清 华 大 学

综 合 论 文 训 练

题目：rCore-Tutorial 页面置换算法与多核同步互斥机制

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：徐奥淳

指导教师：陈　渝　副教授

2022 年 6 月 1 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

本文主要介绍了操作系统中的缺页异常处理，几种页面置换算法的思想，和多核情况下内核态的同步互斥机制，以及它们在 rCore-Tutorial 操作系统上的基本实现。

操作系统给应用程序提供了非常大的虚拟地址空间，但实际上操作系统能够分配使用的物理内存空间不可能有那么大，一种方法是将目前不被使用的物理内存移出并暂存到外部存储设备中，那么用户程序可以访问的虚拟内存空间就可以包括存储设备的巨大容量了。这就是页面置换的基本思路，选择移出哪些物理内存则是页面置换算法决定的。

在多核情况下，会存在多个 CPU 竞争同一资源的问题，操作系统要支持多核就必须要解决内核态的同步互斥问题，实现同步互斥机制。

本文介绍了三种局部算法和两种全局算法的实现，设计了完整的测试，分析了各个算法的测试结果，并阐述了 rCore-Tutorial 操作系统中同步互斥机制的设计和多核支持的实现。

关键词：操作系统；缺页异常；页面置换算法；多核；同步互斥

ABSTRACT

This paper introduces the page fault exception handling in the operating system, the ideas of several page replacement algorithms, the synchronization and mutual exclusion mechanism of the kernel state in the case of multi-core, and their basic implementation on the rCore-Tutorial operating system.

The operating system provides a very large virtual address space for applications, but in fact the physical memory space that the operating system can allocate and use cannot be that large. One method is to move the physical memory that is not currently in use and temporarily store it in an external storage device. Then user programs can access the huge capacity of the storage device. This is the basic idea of page replacement. The choice of which physical memory to remove is determined by the page replacement algorithm.

In the case of multi-core, there will be multiple CPUs competing for the same resource. To support multi-core, the operating system must solve the problem of synchronization and mutual exclusion in kernel mode and realize the synchronization and mutual exclusion mechanism.

This paper introduces the implementation of three local algorithms and two global algorithms, designs a complete test and analyzes the test results of each algorithm. We also expounds the design of the synchronization and mutual exclusion mechanism in the rCore-Tutorial operating system and the realization of multi-core support.

Keywords: OS; page fault; page replacement algorithm; multi-core; Synchronization and Mutex

目　录

主要符号表

OS	操作系统 (Operating System)
PCB	进程控制块 (Process Control Block)
TCB	任务控制块 (Task Control Block)
MMU	内存管理单元 (Memory Management Unit)
RAII	资源获取即初始化 (Resource Acquisition Is Initialization)
PC	指令计数器 (Program Counter)

第 1 章　引言

操作系统（Operating System，OS）是计算机中管理 CPU、内存和各种外设等硬件的系统程序，它为用户的应用程序提供各种底层的支持和服务，还是用户程序和计算机硬件之间的沟通者和中间层。从用户的角度而言，操作系统的设计主要考虑用户的使用方便，而系统的性能和资源利用显得没有那么重要，但从计算机系统的角度而言，所有硬件都与之紧密相关，操作系统需要合理、高效地管理这些资源，这时可以将其看作资源分配器，如何给所有运行在操作系统上的用户程序分配资源是非常重要的。

虚拟内存（virtual memory）是一种管理计算机内存的重要技术，它将用户应用程序的逻辑内存与物理内存分开，使用户认为自己能够使用一个非常大且连续的虚拟地址空间（virtual address space），这种设计具有透明性、高效性和安全性，即用户程序不需要知道硬件上物理内存的细节，也不必要关心操作系统内核实现的策略，同时虚拟内存带来的额外开销较小，并且可以检测和防止用户程序对内核或者其他应用程序进行攻击或其他恶意的行为。

随着虚拟内存管理而来的，是缺页异常（page fault）的出现和操作系统需要对其进行的处理。当用户程序对一个虚拟地址（virtual address）进行访存时，MMU 硬件会试图通过页表（page table）将其转换为物理地址（physical address），转换的过程中检查该页表项（page table entry）的权限是否合法，若在这个过程中出现错误则会触发缺页异常，这时就需要内核的相应处理程序对缺页异常进行适当的处理，使得用户程序能够正常执行下去或者杀死用户程序。其中一种情况下，当用户程序使用的内存超过操作系统实际能够使用的物理内存时，访存指令触发缺页异常后，操作系统需要通过页面置换将暂时不被使用的页面换出到外部存储设备中，再将访问的物理页面放入内存中以成功重新执行该指令。

本文将在已有的 rCore-Tutorial-v3 操作系统上，以教学为主要目的，实现其之前没有的页面置换机制和各种页面置换算法，设计一套完整的测试用例，对机制和算法进行测试并从多个角度分析得到的测试结果。然后，本文将按照 rCore-Tutorial-v3 操作系统的架构一步一步探索其多核版本的实现，使其支持多核下的内核态同步互斥机制，并成功运行在 qemu 环境下。

由于本文的项目完全以 rCore-Tutorial-v3 操作系统为基础，本章将会详细介绍其架构和实现，在此之前需要先简单介绍一些 Rust 语言的特性和优势。

1.1 Rust 语言简介

Rust 语言是一门由 Mozilla 主导开发的系统级的编程语言，有着“安全，并发，实用”的设计准则，主要目的是解决一系列系统问题，特别是需求高安全性和高并发的场景。它的语法与 C/C++ 类似，能提供与 C++ 不相上下的性能，但语义却非常不同，为了提供更好的编译时安全检查，Rust 设计了自己的类型系统，其“trait”直接模仿 Haskell 语言的 type class 概念，并在语法上添加了所有权机制，以便在编译期就检查出一些可能的错误。

1.1.1 所有权

Rust 的所有权系统规定，一个内存上的可变数据，也就是变量，都需要受到所有权的限制，所有值都唯一属于一个自己的属主。对一个变量的不可变的引用可以在同一时间存在多个，而对一个变量的可变引用只能存在一个，并且不可变引用和可变引用不能同时存在。在编译期 Rust 就可以检查代码中所有引用的合法性和安全性，保证 Rust 代码做到内存安全。

1.1.2 安全性

安全性方面主要考虑内存安全和线程安全。

Rust 没有 Java 和 Go 一样的垃圾回收系统，而是用 RAII 的思想来对内存和不同资源进行管理，也提供引用计数来管理变量，让数据或资源与变量的生命周期绑定，当一个资源绑定的变量生命周期结束时资源也被自动释放了。Rust 的引用概念不同于运行时引用计数，而是在编译时保证 Rust 代码里的安全区域中不能存在空指针、悬垂指针以及其他未定义行为。定义的数值初始化只能使用 Rust 提供的一系列固定的形式，并且要求所有定义的变量或输入的变量都已经完成了初始化。

在多线程情境下，多个线程可能会同时对同一数据或资源进行访问，引发资源竞争，想要达到并发安全必须要考虑这个问题。Rust 要求，如果一个变量或资源可能同时被多个线程使用，就需要实现一个叫 Sync 的 Trait，或用该 Trait 实现的锁封装该资源，Sync 的含义就是线程安全的。

1.2 rCore-Tutorial-v3 介绍

rCore-Tutorial-v3 是一个用 Rust 语言实现的以教学为目的的基本功能完备的操作系统，可以运行在 qemu 模拟器环境下和 K210 真机平台上，并且能够享受上述 Rust 语言特性带来的好处。

既然是以教学为目的实现的，rCore-Tutorial-v3 按照易于学习的逻辑分为九个章节（chapter）：第一章，从零搭建最基本的内核执行环境；第二章，实现简单的批处理系统（Batch System）；第三章，实现任务切换和分时多任务系统；第四章，引入地址空间，实现 SV39 多级页表机制；第五章，实现进程管理机制；第六章，引入简易文件系统（easy-fs）并接入内核；第七章，实现进程间通信和 I/O 重定向；第八章，实现线程管理，支持并发；第九章，实现 I/O 设备管理。第一章到第九章对应的项目分支分别为为 ch1-ch9。

跟随 rCore-Tutorial-v3 每一章逐步前进，能够对现代操作系统的实现有一个清楚的基本的了解，这也是该项目的目的，本文的工作是在完成的 rCore-Tutorial-v3 版本上实现页面置换机制和各种页面置换算法，使其支持缺页异常处理，以及从零开始实现每一章对应的多核版本，解决多核情况下内核态的同步互斥问题。

本节将详细讲解 rCore-Tutorial-v3 的代码框架和与本文高度相关的内存管理模块使用的 SV39 多级页表机制。

1.2.1 代码框架

rCore-Tutorial-v3 使用 Rust 语言实现，内核中一共设计了 13 个模块：mm 模块，内存管理模块；trap 模块，实现用户态的中断异常，用户态与内核态的上下文切换；task 模块，实现了进程管理机制和调度算法；sync 模块，线程管理与线程安全；syscall 模块，实现系统调用；sbi 模块，提供 RustSBI 服务；console 模块，实现命令行输出；timer 模块，计时器与时钟中断相关；config 模块，管理操作系统的各种参数和常量；lang_items 模块，实现 Rust 要求的 panic_handler；fs 模块，文件系统；board 模块，平台相关，定义了 qemu 模拟器和 K210 真机上的不同参数；drivers 模块，管理块设备驱动。

同时，rCore-Tutorial-v3 也配置了用户程序的编译环境，使其能够和内核的设计实现同步，使用内核提供的系统调用。上述模块中，内存管理模块与本文关联较为紧密，所以会在下面小结进行详细讲解，在讲解之前先介绍 SV39 多级页表机制。

1.2.2 SV39 多级页表

在介绍 SV39 多级页表机制之前，先简单介绍一下分页式的内存管理。

内核以页为单位对物理内存进行管理，每个用户程序的虚拟地址空间被划分为若干虚拟页面（page），真正的物理内存也被划分为若干物理页帧（frame），它们的大小相同，每个虚拟页面对应到一个物理页帧，虚拟页面上的数据实际上也是存储到物理页帧上。

这样管理虚拟内存自然会带来地址转换的问题，对于每个虚拟地址，访问前操作系统需要找到它对应的物理地址，这个过程就叫地址转换。在分页式内存管理中，为了方便，需要给每个虚拟页面一个编号，叫做虚拟页号（Virtual Page Number，VPN），物理页帧也是一样，叫做物理页号（Physical Page Number，PPN）。每个用户程序使用自己的页表将自己的虚拟页面映射到实际的物理页帧上，MMU 使用页表对内存进行访问。

在 SV39 多级页表机制中，虚拟地址为 39 位，格式如图 1.1[1]所示，物理地址为 56 位，格式如图 1.2[1]，分页管理的每个页面大小为 4KiB，而 4KiB 需要 12 位字节的地址来表示，所以虚拟地址和物理地址的低 12 位都用来表示页面内的位置，叫做页内偏移（Page Offset）。虚拟地址的高 27 位用来表示它的虚拟页号，而物理地址的高 44 位表示它的物理页号。地址转换时以页为单位，所以转换前后地址的低 12 位也就是页内偏移不变，只取虚拟地址的虚拟页号，转换为物理页号，最后与之前的页内偏移组成可以访问的物理地址。

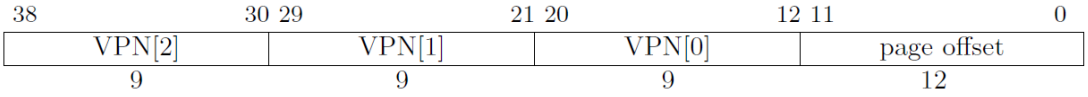

图 1.1　SV39 多级页表的虚拟地址格式与组成

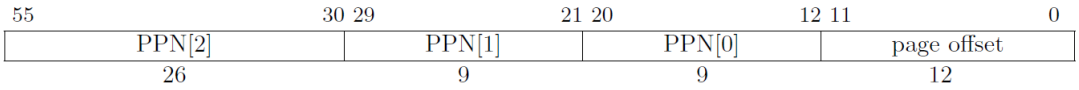

图 1.2　SV39 多级页表的物理地址格式与组成

SV39 多级页表的页表项如图所示 1.3[2]，其为虚拟页号在页表中查询到的结果，[63：54] 为保留位，[53：10] 为物理页号，[7：0] 为标志位，其中，V（Valid）为 1 时页表项合法，R（Read）、W（Write）、X（eXecute）表示索引到该页表项的虚拟页面是否可读、可写、可执行，U（User）表示索引到该页表项的虚拟页面是否可以被 U 特权级下的指令访问，A（Accessed）表示这一位上一次被清理

到现在这段时间内该虚拟页面是否被访问过，D（Dirty）表示这一位上一次被清理到现在这段时间内该虚拟页面是否被修改过。

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

图 1.3　SV39 多级页表的页表项

SV39 多级页表的虚拟页号被分为三级页索引（Page Index），所以它是一个三级页表。从数据结构的角度来看，其为一棵深度为 4 的字典树，每个非叶节点有 $2^9 = 512$ 个子节点，非叶节点的页表项标志位与上述的叶节点页表项标志位不完全相同。当 V=1 且 R、W、X 都为 0 时，该节点为一个合法的页目录表项，即非叶节点，其包含的物理页号指向下一级页表；当 V=1 且 R、W、X 不全为 0 时，该节点为一个合法的页表项，即叶节点，其包含的物理页号即为虚拟页号对应的物理页号。

1.2.3　内存管理模块

rCore-Tutorial-v3 的内存管理模块在 mm 模块中，代码结构如下：

```
mm
├── address.rs
├── frame_allocator.rs
├── heap_allocator.rs
├── memory_set.rs
├── mod.rs
└── page_table.rs
```

其中，address 中实现了虚拟地址、物理地址和虚拟页号、物理页号的结构体，以及地址和相应页号之间的转换，还实现了 VPNRange 结构，保存一段连续的虚拟页号并可以使用迭代器迭代。

frame_allocator 中实现了物理页帧的分配器 StackFrameAllocator 和使用 RAII 思想的 FrameTracker，用分配器分配的物理页帧会被封装进一个 FrameTracker，当其被释放（drop）时会调用类似于析构函数的过程，会将该物理页帧释放掉，分配器收集其用来再次分配。heap_allocator 中实现了使用伙伴系统分配内存的堆分配器，用来作为 Rust 程序所要求的 global_allocator，因为没有了 std 库，操作系统需要自己实现这部分功能。

page_table 中实现了页表项和页表，可以建立 SV39 虚拟页号到物理页号的映射。memory_set 中实现了内核和用户程序用来管理内存的数据结构 MemorySet

和 MapArea，内核和每个用户程序都有自己的一个 MemorySet，整个内核地址空间就由 MemorySet 类型的 KERNEL_SPACE 管理，而用户程序的 MemorySet 实例在对应的 PCB 中，MemorySet 的结构如下：

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

包含一个页表和一个存储若干 MapArea 的向量，内存的组织形式为一段一段连续分配的内存段，每一段内存的虚拟页面由一个 MapArea 管理，这些内存段需要保证互不相交或包含，否则内核将抛出错误。MapArea 的结构如下：

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

其中 vpn_range 为上文提到的 VPNRange 类型，保存分配的连续虚拟页号。map_type 保存这一段内存映射的方式，有恒等映射和使用页表映射两种方式，内核空间中使用的都为恒等映射，而用户程序分配的内存都使用页表映射，这样内核就可以看似直接访问物理地址了。map_perm 保存这一段内存的权限，记录 R、W、X 和 U 这四位。

data_frames 为虚拟页号到一个 FrameTracker 的映射，记录本段内存有哪些虚拟页面是分配了物理页帧的，并用 RAII 的思想对物理页帧进行高效的管理，当一个 MapArea 实例被释放时，其中 data_frames 中所有的 FrameTracker 也会被释放，进而由上文提到的析构函数会将所有分配的物理页帧回收重新利用，这样就不需要特意去回收被释放的物理页帧了，非常方便。

MapArea 上实现了映射一个虚拟页面到物理页帧的方法和释放一个虚拟页面的方法，MemorySet 中实现了分配一段虚拟内存的方法，在我的工作前，该方法会直接把这段虚拟内存对应的所有虚拟页面都直接立刻映射一个物理页帧，不管用户程序是否会使用。

1.3 本文工作

本文基于上述 rCore-Tutorial-v3 的 main 分支，结合其内存管理模块实现了页面置换机制，使操作系统能处理用户态的缺页异常，在此机制上实现了三种局部页面置换算法：FIFO 算法、Clock 算法和改进的 Clock 算法，两种全局页面置换算法：缺页率置换算法和工作集置换算法。

为了测试这五种算法的正确性和性能，本文设计了七个测试用例，使用不同的访存模式对各个算法进行测试，得到了与预期较为符合的测试结果，并对这些测试结果进行了详细分析。

然后，本文从第一章（ch1）开始，实现支持多核的版本，解决内核态的同步互斥问题，实现同步互斥机制，直到第八章（ch8）结束，实现了每一章对应的多核版本（ch1-smp 到 ch8-smp），测试时系统设定为四个核。

第 2 章　页面置换机制

页面置换中的页面指的是物理页帧，置换是指将操作系统能够利用的物理内存中的物理页帧移出到外部存储设备中，或将之前移出的物理页帧从外部存储设备再次移入物理内存。计算机的物理内存是有限的，如果能将有巨大容量的外部存储设备的存储空间也利用起来，那么操作系统将可以真正给用户应用程序提供巨大的虚拟地址空间。

本章将讨论操作系统为什么需要页面置换机制，解释页面置换的出发点，然后从这个出发点，本文将介绍文章设计的页面置换机制的实现。这个实现主要以教学为目的，不考虑系统具体执行的性能，追求简洁、清晰的逻辑，尝试让读者能够高效地理解页面置换机制的实现。

2.1　为什么需要页面置换

前文提到，操作系统给用户程序提供了非常大的虚拟内存空间，但实际可以使用的物理内存却是有限的。为了达到这个目的，操作系统的做法是只将用户程序的一部分常用的数据和代码存到物理内存中，将其他不常用的数据和代码存储到外部存储设备中。当用户程序访问前者时可以直接访问到，而访问后者时则会触发缺页异常，这时操作系统才需要尽快地将用户程序需要的数据重新放入真正的物理内存中，然后重新执行触发异常的访存指令。

在实际应用中，操作系统一定会遇到上述情况，这时如何判断哪些页面是常用的，哪些页面是不常用的，在物理内存不足时将不常用的页换出而将常用的页保留在内存中，是操作系统要仔细考虑的问题，这里就引入了页面置换算法。由于访问外部存储设备的代价比直接访问物理内存大得多，所以一个好的页面置换算法会使用户程序尽可能少地触发缺页异常，也就是尽可能少地访问外部存储设备，得到一个尽可能高的系统执行效率。

2.2　页面置换机制的实现

想要实现页面置换机制，我们不仅仅需要考虑页面置换算法相关的设计实现，还需要先考虑其他机制上的问题：比如当前内存中哪些物理页面可以被换出到外

部存储设备中？不同用户程序的虚拟页面与外部存储设备存储结构（比如硬盘的扇区）的映射关系是怎样的？如何设计数据结构以支持页面置换算法？如何完成物理页面的换入换出操作？

本节将一一讨论这些细节，实现一个可用的页面置换机制。

2.2.1 可以被换出的物理页面

首先比较显然的是，在操作系统内不是所有物理页面都可以被置换到外部存储设备中的，只有被用户虚拟地址空间映射且可以直接被用户应用程序访问的物理页面才能被移出，内核需要使用的那部分内存空间的物理页面则需要一直保留在内存中。

上述结论的原因是操作系统需要运行和访问的是关键的代码和数据，若想保证操作系统整体运行的高效和稳定，就需要将这部分代码和可能使用到的数据保留在物理内存中。可以想象，在操作系统内核代码执行时，如果内核触发了缺页异常，那么操作系统将必须等待很长时间，因为外部存储设备比如硬盘的访问速度比物理内存慢两到三个数量级，整个系统的运行效率将非常低下。更致命的是，如果缺页异常的处理相关的内核代码或数据此时不在物理内存中，操作系统的内核甚至可能会直接崩溃。在 rCore-Tutorial-v3 操作系统的设计中，内核运行在 Supervisor 态，并且内核态的中断异常被禁止了，操作系统不能对 S 态的中断异常做任何处理，所以需要将内核相关的代码和数据保持在物理内存中。

在本文的设计中，操作系统仅将所有用户程序动态申请的内存对应的物理页面视为可交换出去的，这在之后的页面置换算法具体实现中可以看出来。

2.2.2 虚拟磁盘

由于在 qemu 模拟器中，没有真正的模拟硬盘，本文中使用的方法是，将内核空间中静态存储区（static）中的一段内存空间视为硬盘的存储空间，并且按照硬盘的读写方式访问这段内存，将其封装进一个硬盘访问驱动的接口中。实际上硬盘和物理内存的区别在于，前者掉电后数据不易失但访存速度慢，后者掉电后数据易失但访存速度快，但这些区别对于要实现的页面置换机制本身来说并不本质，它们只不过是两块不同的存储空间，数据在这两块空间上传输。并且，操作系统其实并不要求硬盘上的存储空间作为物理内存交换的那部分不易失，因为实际上这部分也被操作系统视为物理内存，而物理内存本身就是易失的。

这样的设计可能有点奇怪，因为经过这一系列操作后，操作系统能使用的物理页面总数其实并没有增加，只是将之前内核可以直接使用的一些内存中的物理

页面移动到我们虚拟磁盘中的内存中了。由于本文是以教学为目的的，文章在这里只考虑页面置换机制的理论逻辑和实现，而不关心操作系统内核处理缺页异常的实际性能，只考虑页面置换算法导致的缺页率不同，而这与外部存储设备是什么、怎么实现没有关系，于是在虚拟磁盘的实现上采取了适当的简化。

本文使用 IDE 来表示虚拟磁盘，IDE（Integrated Drive Electronics）表示一种标准的硬盘接口，但文章中实现的虚拟磁盘和 Integrated Drive Electronics 本身其实并不相关，这个命名是 ucore 的历史遗留，这里借用了这个命名。实现代码中将 IDE 结构放在了 driver 模块下的 ide.rs 中，表示这是虚拟磁盘的驱动。

```
pub const MAX_PAGES: usize = 512;
pub const IDE_SIZE: usize = MAX_PAGES * PAGE_SIZE;

#[repr(align(4096))]
struct IDE {
    pub data: [u8; IDE_SIZE],
}
```

在上面这段代码中，MAX_PAGES 为设定的虚拟磁盘总的可以载入的物理页面数，将其和页面大小 PAGE_SIZE 相乘后得到虚拟磁盘的总大小 IDE_SIZE。为了实现的方便，本文将虚拟磁盘的一个磁盘扇区设置成了刚好一个物理页面的大小，并且将 IDE 结构使用 4096 字节对齐，这样做的好处在于将物理页面移入和移出虚拟磁盘时可以非常方便地进行操作：

```
pub fn ide_read(idx: usize, dst: &mut [u8]) -> usize {
    if !ide_valid(idx) {
        return 1;
    }
    let base = idx * PAGE_SIZE;
    unsafe {
        let ide_ptr = &IDE.data[base..(base+PAGE_SIZE)];
        dst.copy_from_slice(ide_ptr);
    }
    0
}
pub fn ide_write(idx: usize, src: &[u8]) -> usize {
    if !ide_valid(idx) {
        return 1;
    }
    let base = idx * PAGE_SIZE;
    unsafe {
        let ide_ptr = &mut IDE.data[base..(base+PAGE_SIZE)];
        ide_ptr.copy_from_slice(src);
```

```
    }
    0
}
```

以读取虚拟磁盘数据为例，方法 ide_read 的参数中 idx 为虚拟磁盘的索引，dst 为读取出的数据存放的位置，在判断 idx 合法没有超过 MAX_PAGES 后，直接将虚拟磁盘第 idx 个扇区中的数据拷贝到 dst 中即可。

2.2.3 虚拟页面与虚拟磁盘上的扇区之间的映射关系

当一个物理页面被置换到了虚拟磁盘上后，操作系统需要以某种方式记录映射该物理页面的虚拟页面和虚拟磁盘上对应数据的映射关系。一般来说，rCore 可以充分利用每个用户程序页表的页表项来记录这个关系，在上一章 SV39 多级页表的介绍中提到，虚拟页面映射的每一个页表项都保存了映射到的物理页号和该页的权限、访存信息，MMU 硬件会根据虚拟地址中的虚拟页号，在页表中将其翻译为映射的物理页号，最后找到物理地址。而当操作系统提供巨大的虚拟地址空间时，页表项可以提供新的帮助，但本文使用了一个数据结构来保存这种映射关系。

这里先理清一下页面置换的整个流程。考虑这样的情况，一个用户程序申请的虚拟页面对应的物理页面在之前某个时刻被操作系统换出到了虚拟磁盘中，但这时该用户程序想要再次访问该虚拟页面中的数据。

在之前某个时刻操作系统将该物理页面换出到虚拟磁盘中时，用户程序的页表中该虚拟页面对应的页表项存在位修改为 0，并需要保存关键的虚拟页面与虚拟磁盘扇区的映射关系，之后需要换入该物理页面数据时需要用到。在这个时刻，该用户程序想要再次访问该虚拟页面中的数据，MMU 硬件会将虚拟地址翻译为物理地址。在这个过程中会查看页表中查找到的该页表项的存在位，然后发现之前已经被操作系统置零了，说明该虚拟页面的数据实际上不在物理内存中，这时会触发产生缺页异常，操作系统需要将该虚拟页面的数据从虚拟磁盘中取出放回到重新分配的物理页面中。

在本文的实现中，操作系统使用 IdeManager 来管理虚拟磁盘的读写，并保存用户程序申请内存的物理页面是否被换出到磁盘上：

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

```
    map: BTreeMap<(usize, VirtPageNum), usize>,
}
```

其中 current、end 和 recycled 类似于 frame_allocator 的实现，用来分配磁盘的扇区，而 map 用来记录 (token, vpn) 到磁盘扇区编号的映射，其中 token 为用户程序的标识，vpn 为将数据存到虚拟磁盘的虚拟页面的页号，这样操作系统就可以查询每个用户程序的每个虚拟页面的数据是否在虚拟磁盘中，缺页异常时是否需要被重新换入了。

IdeManager 提供了 swap_in 和 swap_out 方法用来将用户程序的虚拟页面数据移入或移出。

2.2.4 缺页异常处理

本小节将讲述如何具体实现内核的缺页异常处理，所有的缺页异常处理代码都在方法 do_pgfault 的实现中，上一小节的 IdeManager 和缺页异常处理方法都被放在了 mm 内存管理模块下的 vmm.rs 中。

首先讨论用户程序触发缺页异常的情况。比如，在用户程序的执行访存指令时由于修改只读页或执行不可执行的数据等原因使处理器无法访问对应的物理页面，MMU 不能成功将虚拟地址转换到物理地址时，处理器会产生一次页访问异常，然后跳转到操作系统的缺页异常处理。当缺页异常处理结束后，经过 trap 回到触发异常的指令并再次执行，用户程序就可以继续执行下去了。

产生缺页异常异常的原因主要有：对目标物理页帧的访问权限不合法；目标物理页帧不存在，即页表项为 0，该虚拟地址没有分配物理页帧；目标物理页帧不在物理内存中，页表项非空，但存在标志位为零，目前数据在外部存储设备中。本文的处理是在换出物理页面数据时直接将该映射在页表中取消，第三种情况会变成第二种情况，并在上一小节的 IdeManager 中查询该虚拟页面是否有数据在虚拟磁盘中。

2.2.4.1 页异常处理函数的实现

接下来介绍缺页异常处理函数 do_pgfault 的实现：

```
pub fn do_pgfault(addr: usize, flag: usize) -> bool
```

其中 addr 为当前触发缺页异常的指令访问的虚拟地址，flag 为 0、1、2 记录该缺页异常是 LoadPageFault、StorePageFault 还是 InstructionPageFault。

首先获取进程的信息，得到该进程的 PCB，通过 PCB 得到用户程序的 token 和 memory_set，将触发异常的指令访存的虚拟地址 addr 转换为虚拟页号 vpn，如下面代码所示：

```
let process = current_process();
println!("[kernel] PAGE FAULT: pid {}", process.pid.0);
let mut pcb = process.inner_exclusive_access();
let token = pcb.get_user_token();
let memory_set = &mut pcb.memory_set;
let va: VirtAddr = addr.into();
let vpn: VirtPageNum = va.floor();
println!("[kernel] PAGE FAULT: addr:{} vpn:{}", addr, vpn.0);
```

然后，利用 memory_set 中的页表 page_table 得到要查询的页表项，查看其是否存在、是否有效来判断该对应的物理页面是否存在且有效，若是则此时的情况一定是访存指令不满足访问权限导致的缺页异常，因为这时物理页面还在内存中，如下面代码所示：

```
if let Some(pte) = memory_set.page_table.translate(vpn) {
    if pte.is_valid() {
        if !pte.readable() && flag==0 {
            println!("[kernel] PAGE FAULT: Frame not readable
  .");
            return false;
        }
        if !pte.writable() && flag==1 {
            println!("[kernel] PAGE FAULT: Frame not writable
  .");
            return false;
        }
        if !pte.executable() && flag==2 {
            println!("[kernel] PAGE FAULT: Frame not
  executable.");
            return false;
        }
    }
}
```

返回 false 表示缺页异常处理失败。然后缺页异常的处理分为两个部分，局部页面置换算法和全局页面置换算法：

```
if PRA_IS_LOCAL {
    local_pra(memory_set, vpn, token)
}
```

```
else {
    global_pra(memory_set, vpn, token)
}
```

2.2.4.2 局部页面置换部分的处理

在局部页面置换算法的处理中，接下来在当前用户程序的 memory_set 的内存段 areas 中查看是否包含触发异常的 vpn ，若不包含则说明该用户程序访问了一个无效的虚拟地址，直接返回错误即可，若包含则说明该用户程序之前是申请了包含该虚拟地址的内存，但内核还未为其分配物理页面，或分配后已经被交换到磁盘中暂时失效了。

```
for i in 0..memory_set.areas.len() {
    if vpn >= memory_set.areas[i].vpn_range.get_start() && vpn
   < memory_set.areas[i].vpn_range.get_end() {
        ...
    }
}
```

直接判断 vpn 是否在某一个连续内存段 area 的 vpn_range 中即可。此前，当用户在 memory_set 中申请一段内存时，若可以先不分配则不分配，只将其记录到 areas 当中：

```
fn push(&mut self, mut map_area: MapArea, data: Option<&[u8]>)
   {
    if let Some(data) = data {
        map_area.map(&mut self.page_table);
        map_area.copy_data(&mut self.page_table, data);
    }
    self.areas.push(map_area);
}
```

这里 data 为 Option<&[u8]> 类型，只有当有需要复制的数据时，也就是用 Some() 成功取出数据时，才使用 map_area 的 map 方法映射这一段虚拟地址并复制数据，否则直接将该 map_area 保存在 areas 当中即可。

回到缺页异常的处理，若触发异常的 vpn 包含在 memory_set 的 areas 中，则说明内核还未为其分配物理页面，或分配后已经被交换到虚拟磁盘中暂时失效了，但不管怎样我们都需要分配一个物理页面给它使用，如下面代码所示：

```
if let Some(frame) = frame_alloc() { // enough frame
```

```
    ppn = frame.ppn;
    memory_set.areas[i].data_frames.insert(vpn, frame);
    println!("[kernel] PAGE FAULT: Frame enough, allocating
  ppn:{} frame.", ppn.0);
}
else {  // frame not enough, need to swap out a frame
    ppn = memory_set.get_next_frame().unwrap();
    let data_old = ppn.get_bytes_array();
    let mut p2v_map = P2V_MAP.exclusive_access();
    let vpn_old = *(p2v_map.get(&ppn).unwrap());
    ide_manager.swap_in(token, vpn_old, data_old);
    for j in 0..memory_set.areas.len() {
        if vpn_old >= memory_set.areas[j].vpn_range.get_start
  () && vpn_old < memory_set.areas[j].vpn_range.get_end() {
            memory_set.areas[j].unmap_one(&mut memory_set.
  page_table, vpn_old);
        }
    }
    p2v_map.remove(&ppn);
    println!("[kernel] PAGE FAULT: (local) Frame not enough,
  swapping out ppn:{} frame.", ppn.0);

    let frame = frame_alloc().unwrap();
    memory_set.areas[i].data_frames.insert(vpn, frame);
}
if ide_manager.check(token, vpn) {
    let data = ppn.get_bytes_array();
    ide_manager.swap_out(token, vpn, data);
    println!("[kernel] PAGE FAULT: (local) Swapping in vpn:{}
  ppn:{} frame.", vpn.0, ppn.0);
}
```

此时需要先查看物理页面是否足够，使用 frame_alloc 分配物理页面，足够时可以直接成功分配，主要考虑物理页面不足需要换出的情况。ppn = memory_set.get_next_frame().unwrap() 获得要换出的物理页面对应的 PPN，根据不同算法这里获得的页面不同，get_next_frame 方法相关的内容第三章会详细讲解。然后将该物理页面的数据（data_old）通过 ide_manager 的 swap_in 方法复制记录到磁盘中，并且将其从页表中删除，最后再将该物理页面分配给触发异常的虚拟地址。

其中 P2V_MAP 记录每个被分配的物理页面对应的虚拟页面的页号，因为需要找到对应的虚拟页号才能将这个映射从页表中删除。删除时需要先在所有保存的连续内存段 areas 中找到该虚拟页号所在的那一段，然后在找到的内存段中取消该虚拟页面的映射。ide_manager 的 check 方法可以查询用户程序的该虚拟页面的数据是否在虚拟磁盘中，如果该虚拟地址之前被分配过物理页面并被换出，操

作系统需要将原来的数据从虚拟磁盘中拿出来放到新分配的物理页面中，这里利用了 ide_manager 的 swap_out 方法。

最后建立新的映射：

```
if !frame_check() {
    let ppn = memory_set.get_next_frame().unwrap();
    ...
}
println!("[kernel] PAGE FAULT: (local) Mapping vpn:{} to ppn
  :{}.", vpn.0, ppn.0);
memory_set.page_table.map(vpn, ppn, memory_set.areas[i].
  get_flag_bits());
P2V_MAP.exclusive_access().insert(ppn, vpn);
memory_set.insert_frame(ppn);
```

由于建立映射时页表可能需要新的物理页面，所以我们先用 frame_check 检查一下物理页面是否还有剩余，若没有则提前换出一个，换出的过程和之前的处理一样，保证页表可以正常工作，然后在用户程序的页表中建立新的映射，并同时更新 P2V_MAP 中的记录。就像 2.2.1 节提到的那样，memory_set 的 insert_frame 方法用于记录页面置换算法可以换出的物理页面。

2.2.4.3 全局页面置换部分的处理

全局部分一开始和局部页面置换部分的处理相同，先在当前用户程序的 memory_set 的 areas 中查看是否包含触发异常的 vpn ，若包含则进行进一步处理。

在进一步处理中，首先进行全局页面置换算法的预处理，两个页面置换算法的处理是不同的，在下一章中会详细讲解，如下：

```
GFM.exclusive_access().work(memory_set, token);
```

由于是全局页面置换算法，参数设置合理的情况下，这时保证一定有空余的物理页面，直接分配即可。同样地，如果该虚拟地址之前被分配过物理页面并被换出，我们需要将原来的数据拿出来放到新分配的物理页面中：

```
let frame = frame_alloc().unwrap();
ppn = frame.ppn;
memory_set.areas[i].data_frames.insert(vpn, frame);

if IDE_MANAGER.exclusive_access().check(token, vpn) {
    let data = ppn.get_bytes_array();
    IDE_MANAGER.exclusive_access().swap_out(token, vpn, data);
```

```
    println!("[kernel] PAGE FAULT: (global) Swapping in vpn:{}
   ppn:{} frame.", vpn.0, ppn.0);
}
```

和局部部分一样，最后建立新的映射：

```
memory_set.page_table.map(vpn, ppn, memory_set.areas[i].
  get_flag_bits());
P2V_MAP.exclusive_access().insert(ppn, vpn);
memory_set.insert_frame(ppn);
```

到这里缺页异常就处理完毕了，中断返回后重新执行该指令时可以正常执行。

第 3 章　页面置换算法

本章将详细讲解如何在第 2 章中实现的页面置换机制上实现一个一个的页面置换算法，先介绍各个页面置换算法会使用到的相关数据结构。

3.1　相关数据结构

在 mm 模块中的 frame_manager.rs 中，本文设计了 ClockQue、LocalFrameManager 和 GlobalFrameManager 等协助完成页面置换算法的类，他们的定义和作用如下：

ClockQue，管理 Clock 和改进 Clock 页面置换算法中的物理页面循环队列，在触发缺页异常时按照算法的定义弹出下一个将被置换的物理页面。ppns 为记录的物理页面循环队列，ptr 为算法当前的指针位置，类上定义了 push 和 pop 方法，在下一节将详细介绍。

```
struct ClockQue {
    ppns: Vec<PhysPageNum>,
    ptr: usize,
}
```

LocalFrameManager，局部页面置换算法的总管理器，保存使用的页面置换算法、FIFO 的队列、Clock 和改进 Clock 的队列，以及协助全局页面置换算法的 global_ppns，它用来记录每个用户程序已经分配的物理页面的页号。每个 MemorySet 类的实例也就是每个用户程序的 memory_set 中都有一个 LocalFrameManager 实例用来进行局部页面管理，类上定义了 get_next_frame 和 insert_frame 方法，用来得到下一个可以被置换的物理页面和插入一个新的物理页面，供 MemorySet 进行调用。

```
pub struct LocalFrameManager {
    used_pra: PRA,
    fifo_que: Queue<PhysPageNum>,
    clock_que: ClockQue,
    global_ppns: Vec<PhysPageNum>,
}
```

GlobalFrameManager，全局页面置换算法的总管理器，保存使用的页面置换

算法，以及两个算法需要用到的变量 t_last 和 idx，在之后相应的算法详细描述中会对它们进行解释。类上定义了三个方法，分别为 pff_work、check_workingset 和 workingset_work 方法，其中第一个为缺页率置换算法相关的方法，后两个为工作集置换算法相关的方法。

```
pub struct GlobalFrameManager {
    used_pra: PRA,
    t\_last: usize,
    idx: usize,
}
```

3.2 局部页面置换算法

一般来说，局部页面置换算法有以下几种：先进先出（First In First Out, FIFO）算法，最久未使用（least recently used,，LRU）算法，时钟（Clock）算法，改进的时钟（Enhanced Clock）算法。本文实现了以上除 LRU 之外的算法，LRU 算法在现有的条件下很难实现，就不对其做过多介绍了。

3.2.1 先进先出（FIFO）算法

顾名思义，先进先出算法会给每一个用户程序单独维护一个队列，该用户程序分配的物理页面会依次从队尾加入队列，每次需要置换物理页面时从队首拿出一个页面换出。

FIFO 算法实现上很简单，但只有在用户访存非常规律时才具有较好的性能，比如线性访存，而在其他更一般的情况下效率不高，因为它对局部性不够敏感，而大多数程序都具有良好的局部性，那些经常被访问的物理页面可能很早就进入队列，而算法却因为这个原因将它们置换出了内存。另一个不好的地方是，FIFO 算法会出现 Belady 现象，在某些情况下，增加内核可以使用的物理页面反而会使用户程序触发缺页异常的次数增多。

FIFO 算法在本文实现的页面置换框架下的实现也非常简单，只需要给每一个用户程序记录一个物理页面队列即可，在用户访问申请的内存并触发缺页异常分配物理页面时将对应 PPN 加入队列的队尾，在需要换出物理页面时将队首的物理页面换出。在上面提到的 LocalFrameManager 的 get_next_frame 和 insert_frame 方法中，实现了该队列：

```
// LocalFrameManager 的 get_next_frame 方法中
match self.used_pra {
    PRA::FIFO => {
        self.fifo_que.pop()
    }
    ...
}

// LocalFrameManager 的 insert_frame 方法中
match self.used_pra {
    PRA::FIFO => {
        self.fifo_que.push(ppn)
    }
    ...
}
```

3.2.2 时钟（Clock）算法

时钟（Clock）算法是 LRU 算法的一种近似实现。此算法将所有可置换的物理页面加入一个循环队列，形状为环形，很像一个时钟，这也是该算法名字的由来。算法设置一个指针，先将该指针指向最早进入链表的物理页面，并且要求用户程序的页表在每个物理页面对应的页表项中设置一位访问（access）位，表示此页表项对应的页当前是否被访问过。

当某个物理页面被访问时，处理器中的 MMU 硬件会自动把对应页表项的访问位置 1，操作系统只需要知道这个事实即可。当时钟算法需要将某个物理页面弹出时，会从当前指针处在环形链表中依次向下一个页面移动，每次查看当前物理页面对应页表项的访问位是否为 0，若是则弹出该页面，否则将访问位置为 0 后移动到下一个页面，直到算法找到可以弹出的页面为止。该算法使用了 LRU 的思想，虽然不能完全达到 LRU 的效果，但易于实现、开销少，并且需要相应的硬件支持，可以在用户访问页面时修改访问位。

在本文的实现上，与 FIFO 页面置换算法相同，内核在 LocalFrameManager 中插入和获取置换的页面，但使用的是 clock_que。

```
// LocalFrameManager 的 get_next_frame 方法中
self.clock_que.pop(page_table)

// LocalFrameManager 的 insert_frame 方法中
self.clock_que.push(ppn)
```

在 ClockQue 类中，上述 push 和 pop 方法如下：

```
pub fn push(&mut self, ppn: PhysPageNum) {
    self.ppns.push(ppn);
}

pub fn pop(&mut self, page_table: &mut PageTable) -> Option<
  PhysPageNum> {
    loop {
        let ppn = self.ppns[self.ptr];
        let vpn = *(P2V_MAP.exclusive_access().get(&ppn).
  unwrap());
        let pte = page_table.find_pte(vpn).unwrap();
        if !pte.is_valid() {
            panic!("[kernel] PAGE FAULT: (local) Pte not valid
   in PRA Clock pop.");
        }
        if !pte.accessed() {
            self.ppns.remove(self.ptr);
            if self.ptr == self.ppns.len() {
                self.ptr = 0;
            }
            return Some(ppn);
        }
        pte.change_access();
        self.inc();
    }
}
```

push 就是简单的将新的物理页面号加入队尾，pop 相对复杂一些，需要循环寻找替换出的物理页面直到找到为止，每次循环中先用 P2V_MAP 查询物理页面对应的虚拟页面，然后用该虚拟页面的页号在页表中查看该物理页面的页表项的 access 位，若为 0 则选择该物理页面弹出，否则将其 access 位置零后查看循环队列的下一位。

3.2.3 改进的时钟（Enhanced Clock）算法

在时钟算法中，选择一个物理页面换出时只考虑了该页面最近是否被访问过，但在实际情况中，还需要考虑最近是否修改过将要换出的物理页面。被修改过的页面需要先写回外部存储设备，这样的代价比换出没有修改过的页面大很多，所以应该尽量先换出未被修改过的页面。改进的时钟置换算法相比于改进前的算法，将可以换出的页面最近的修改情况加入了考虑。该算法首先寻找最近未使用的页，并且在这些页中尽量寻找最近在内存中停留的时间内没有被修改过的页来置换出去。

这要求用户程序的页表在每个物理页面对应的页表项中除了访问位以外再

设置一位修改（dirty）位，表示其当前是否被修改过。当某个物理页面被访问时，处理器中的 MMU 硬件会自动把对应页表项的访问位置 1，而当某个物理页面被修改时，处理器中的 MMU 硬件会自动把对应页表项的修改位置 1。

当时钟算法需要将某个物理页面弹出时，会从当前指针处在环形链表中依次向下一个页面移动，每次查看当前物理页面对应页表项的修改位和访问位，若均为零则弹出该页面，否则两个标志位至少有一个为 1，若修改位为 1 则将其置 0 后移动到下一个页面，若修改位已经为 0 则将访问位置 0 后移动到下一个页面，直到算法找到可以弹出的页面为止。该算法与之前的时钟算法相比，可以更好地减少缺页异常的次数，但为了找到一个更适合换出的物理页面，算法可能需要更多的访问循环队列（环形链表），从而增加了算法本身的开销。

在具体实现上，改进的 Clock 和 Clock 页面置换算法相似，只有 pop 操作不同，改进后的如下：

```
pub fn pop_improved(&mut self, page_table: &mut PageTable) ->
  Option<PhysPageNum> {
    loop {
        ...
        if !pte.accessed() && !pte.dirty() {
            ...
            return Some(ppn);
        }
        if pte.accessed() {
            pte.change_access();
        } else {
            pte.change_dirty();
        }
        self.inc();
    }
}
```

其中省略号同 Clock 算法，不同的是每次循环中先检查该物理页面的页表项 access 和 dirty，若都为 0 则选择该页面弹出，否则若 access 为 1，则将其置零并循环下一位，若 access 为 0 而 dirty 为 1，则将 dirty 置零并循环下一位，直到找到为止。

3.3 全局页面置换算法

上述局部页面置换算法没有设计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。下面介绍的全局置换策略就具有这样的特征。

全局页面置换算法有缺页率置换算法和工作集置换算法。

3.3.1 缺页率置换算法

在上面的各种置换算法中，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，而置换算法的目标是减少缺页次数或缺页率。缺页率置换算法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。

缺页率的定义为：缺页率（page fault rate）= 缺页次数 / 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 Ts 表示适中的缺页率。

这样，缺页率置换算法的基本思路就是：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>Ts，则将在 [t_last,t_current] 这段时间内没有被访问过的页面全部置换出去，然后将缺失的页加入工作集；如果 t_current−t_last≤Ts，则只将缺失的页加入工作集。

在上述思路描述中，t_current－t_last>Ts 表示缺页率低了，通过置换出在 [t_last,t_current] 时间内没有被访问过的页，来将任务的常驻集减小，而 t_current−t_last≤Ts 表示缺页率高了，需要增加任务的常驻集，并且物理页面是否被引用是根据用户程序访问的物理页面对应的页表项的存在位信息来判断的。

在本文的具体实现中，缺页率算法设置一个常数 PFF_T 表示两次缺页异常间隔的阈值，如果两次缺页异常的间隔大于这个阈值，则换出所有在这段时间内不活跃的物理页面，如果小于等于这个阈值，则将所有页面的访问位清除以便下次判断每个页面是否在这次和下次之间的时间内活跃。

此算法的实现在 GlobalFrameManager 的 pff_work 方法中，首先是 t_current-t_last>PFF_T 的情况：

```
for i in 0..task_manager.ready_queue.len() {
    let process = task_manager.ready_queue[i].process.upgrade
```

```
  ().unwrap();
    let mut pcb = process.inner_exclusive_access();
    let token = pcb.get_user_token();
    let memory_set = &mut pcb.memory_set;
    for j in (0..memory_set.frame_manager.global_ppns.len()).
  rev() {
        let ppn = memory_set.frame_manager.global_ppns[j];
        ...
    }
}
for i in (0..memory_set_.frame_manager.global_ppns.len()).rev
  () {
    let ppn = memory_set_.frame_manager.global_ppns[i];
    ...
}
```

这种情况需要换出所有任务的不活跃物理页面，第一个循环是处理在 task_manager 中的所有任务，当前任务不在 task_manager 的 ready_queue 中，从每个任务的 PCB 获取到用户程序的 memory_set，进而可以遍历所有该用户程序可以被换出的物理页面（在 3.1 节中提到的 LocalFrameManager 的 global_ppns 中），第二个循环是处理当前触发缺页异常的任务。

上面代码的省略号中，获取到物理页面的页号 (ppn) 后进行如下处理：

```
let data_old = ppn.get_bytes_array();
let mut p2v_map = P2V_MAP.exclusive_access();
let vpn = *(p2v_map.get(&ppn).unwrap());
if let Some(pte) = memory_set.page_table.translate(vpn) {
    if pte.is_valid() && !pte.accessed() {
        IDE_MANAGER.exclusive_access().swap_in(token, vpn,
  data_old);
        for k in 0..memory_set.areas.len() {
            if vpn >= memory_set.areas[k].vpn_range.get_start
  () && vpn < memory_set.areas[k].vpn_range.get_end() {
                memory_set.areas[k].unmap_one(&mut memory_set.
  page_table, vpn);
            }
        }
        p2v_map.remove(&ppn);
        memory_set.frame_manager.global_ppns.remove(j);
    }
}
```

用上文用到过的流程查看该物理页面的页表项的访问位，若访问位为 0 则需要换出，将其保存到磁盘内，然后取消 memory_set 中该物理页面的映射。

然后是 t_current-t_last<=PFF_T 的情况，则将所有物理页面的访问位清除：

```
for i in 0..task_manager.ready_queue.len() {
    let process = task_manager.ready_queue[i].process.upgrade
  ().unwrap();
    let mut pcb = process.inner_exclusive_access();
    let memory_set = &mut pcb.memory_set;
    for j in 0..memory_set.frame_manager.global_ppns.len() {
        let ppn = memory_set.frame_manager.global_ppns[j];
        ...
    }
}
for i in 0..memory_set_.frame_manager.global_ppns.len() {
    let ppn = memory_set_.frame_manager.global_ppns[i];
    ...
}
```

和 t_current-t_last>PFF_T 的过程类似，遍历所有用户程序可以被换出的物理页面，获取到物理页面的页号 (ppn) 后进行如下处理：

```
let p2v_map = P2V_MAP.exclusive_access();
let vpn = *(p2v_map.get(&ppn).unwrap());
if let Some(pte) = memory_set.page_table.find_pte(vpn) {
    if pte.is_valid() && pte.accessed() {
        pte.change_access();
    }
}
```

只需要将该物理页面的页表项的 access 位置零。上一章缺页异常的全局页面置换算法的预处理会调用该 pff_work 方法，参数 PFF_T 设置合理的情况下，算法保证在处理后有空余的物理页面。

3.3.2 工作集置换算法

工作集置换算法相较于其他算法实现更复杂，因为操作系统内核无法准确获取用户程序每次访存的位置，也就无法获取准确的工作集，但我们可以利用定期的时钟中断和每个页表项的引用位来尽量模拟工作集置换算法。

例如，假设工作集大小 Δ 为最近 20000 次访存，并且每两次时钟中断之间平均会有 4000 次访存，当一次时钟中断触发时，保存所有页面的引用位后全部清除，之后某时刻发生缺页异常时，我们就可以检查当前的访问位和保存在内存中的访问位 n 个位共 n+1 个位，这些位可以确定过去的 4000*(n+1) 次访存中该页面是否被使用过，若被访问过，这 n+1 位中至少有一位会被置一，若没有被访问过，那么这 n+1 位会被全部置零。至少有一位访问位为一的页面被视为在工作集中。

本文对工作集算法的近似类似于时钟算法对于 LRU 算法的近似。LRU 算法需要操作系统知道用户程序访存页面的准确顺序，以便弹出最久未使用的那个页面，而时钟算法则利用页表项的 access 位来通过时钟轮转的方法近似 LRU 算法。工作集置换算法主要需要知道前面一段时间内所有被访存的页面，操作系统无法知道准确的过去一段时间被访存的页面，但也可以通过页表项的 access 位来间接得到这个信息，但由上述本文算法的描述，这段时间需要以时钟中断为基本单位，也就是过去一段时间被访存的页面需要以一个时钟中断平均发生的访存页面数为单位。

考虑算法的有效性，工作集置换算法的正确性和有效性和工作集的大小设置没有本质关系，本文算法可以将工作集大小设置为某一个值的倍数，并不影响工作集算法其他部分的执行，所以有效性可以保证。但该算法不能将工作集的大小随意设置，所以只能是对工作集置换算法的一个近似。

在本文的具体实现中，内核在每次时钟中断的处理时，先调用 GlobalFrameManager 的 check_workingset 方法，复制并清除所有用户程序的所有页面的引用位：

```
Trap::Interrupt(Interrupt::SupervisorTimer) => {
    check_workingset();
    set_next_trigger();
    check_timer();
    suspend_current_and_run_next();
}
```

在 check_workingset 的实现中，也是遍历所有用户程序分配的可换出的物理页面，获取其物理页号：

```
let mut pte_recorder = PTE_RECORDER.exclusive_access();
pte_recorder[self.idx].clear();
let task_manager = TASK_MANAGER.exclusive_access();
for i in 0..task_manager.ready_queue.len() {
    let process = task_manager.ready_queue[i].process.upgrade
  ().unwrap();
    let mut pcb = process.inner_exclusive_access();
    let token = pcb.get_user_token();
    let memory_set = &mut pcb.memory_set;
    for j in (0..memory_set.frame_manager.global_ppns.len()).
  rev() {
        let ppn = memory_set.frame_manager.global_ppns[j];
        ...
    }
```

```
}

let process = current_process();
let mut pcb = process.inner_exclusive_access();
let token = pcb.get_user_token();
let memory_set = &mut pcb.memory_set;
for j in (0..memory_set.frame_manager.global_ppns.len()).rev()
   {
    let ppn = memory_set.frame_manager.global_ppns[j];
    ...
}
self.idx = (self.idx + 1) % WORKINGSET_DELTA_NUM;
```

其中，WORKINGSET_DELTA_NUM 为近似工作集算法时设置的时钟中断数 n，在最近 n 次时钟中断时都要复制记录下所有页面的引用位，PTE_RECORDER 为记录引用位的一个映射循环队列，idx 为当前循环指标。上面代码省略号中，获取到物理页面的页号后，进行以下处理：

```
let p2v_map = P2V_MAP.exclusive_access();
let vpn = *(p2v_map.get(&ppn).unwrap());
if let Some(pte) = memory_set.page_table.find_pte(vpn) {
    if pte.is_valid() {
        pte_recorder[self.idx].insert((token, vpn), pte.
  accessed());
        if pte.accessed() {
            pte.change_access();
        }
    }
}
```

先查询对应的虚拟页号，若页表项有效，则建立用户程序 token 和虚拟页号到当前页表项 access 位的映射，相当于将其记录下来，然后清除该引用位。

然后是 workingset_work 方法，同样地，先遍历所有用户程序分配的物理页面，获取其物理页号，然后进行以下处理：

```
let mut p2v_map = P2V_MAP.exclusive_access();
let vpn = *(p2v_map.get(&ppn).unwrap());
let mut flag = false;
for k in 0..WORKINGSET_DELTA_NUM {
    if let Some(result) = pte_recorder[k].get(&(token, vpn)) {
        if *result == true {
            flag = true;
            break;
        }
    }
```

```
}
if let Some(pte) = memory_set.page_table.find_pte(vpn) {
    if pte.is_valid() && pte.accessed() {
        flag = true;
    }
}
if !flag {
    let data_old = ppn.get_bytes_array();
    IDE_MANAGER.exclusive_access().swap_in(token, vpn,
  data_old);
    for k in 0..memory_set.areas.len() {
        if vpn >= memory_set.areas[k].vpn_range.get_start() &&
   vpn < memory_set.areas[k].vpn_range.get_end() {
            memory_set.areas[k].unmap_one(&mut memory_set.
  page_table, vpn);
        }
    }
    p2v_map.remove(&ppn);
    memory_set.frame_manager.global_ppns.remove(j);
    println!("[kernel] PAGE FAULT: Swapping out ppn:{} frame
  .", ppn.0);
}
```

先查询对应的虚拟页号，然后在 PTE_RECORDER 记录中查询之前 WORKINGSET_DELTA_NUM 个时钟中断中以及这一次时钟中断中是否有访问位为 1，也就是 flag 的含义，若这些时钟中断之中都没有访问位为 1，则该物理页面需要被换出，通过 IDE_MANAGER 将其数据保存至磁盘中，然后取消其在 memeory_set 中的映射即可。

同缺页率算法一样，上一章缺页异常的全局页面置换算法的预处理会调用该方法，在参数 WORKINGSET_DELTA_NUM 设置合理的情况下，在处理后保证有空余的物理页面。

第 4 章　页面置换算法的测试与分析

本章将先对页面置换机制的正确性进行一个简单的测试，然后介绍本文设计的七个完整的测试用例，最后分析第三章中实现的各个算法在这七个测试用例上运行的测试结果。

4.1　简单测试

初步测试使用 rCore-Tutorial-v3 的 ch4 分支中的测例进行测试，测试算法为 FIFO 算法，目的为检验页面置换机制的正确性。首先可以看到操作系统启动后可以使用的物理页面数量为 454，如图 4.1所示。

图 4.1　rCore-Tutorial-v3 启动信息

对于下面代码的 test2_mmap0，将其中 len 设置得较大，使得其分配的内存大于操作系统可以使用的总的物理页面内存，就可以触发物理页面的换入和换出：

```
#[no_mangle]
fn main() -> i32 {
    let start: usize = 0x10000000;
    let len: usize = 4096 * 800;
    let prot: usize = 3;
    assert_eq!(0, mmap(start, len, prot));
    for i in start..(start + len) {
        let addr: *mut u8 = i as *mut u8;
        unsafe {
            *addr = i as u8;
        }
    }
    for i in start..(start + len) {
        let addr: *mut u8 = i as *mut u8;
        unsafe {
            assert_eq!(*addr, i as u8);
        }
    }
    println!("Test 04_1 OK!");
    0
}
```

将其设置为 4096*800，即需要 800 个物理页面，此时操作系统可用的物理页面不足，需要换入和换出物理页面，会触发 StorePageFault 和 LoadPageFault，并通过测试，如图 4.2所示。

```
[kernel] !!!!!!!!!!!!!!!!!!!!!!LoadPageFault!!!!!!!!!!!!!!!!!!!!!!
[kernel] PAGE FAULT: addr:271699968 vpn:66333
[kernel] PAGE FAULT: Frame enough, allocating ppn:526101 frame.
[kernel] PAGE FAULT: Swapping in vpn:66333 ppn:526101 frame.
[kernel] PAGE FAULT: Swapping out ppn:526102 frame.
[kernel] PAGE FAULT: Mapping vpn:66333 to ppn:526101.
[kernel] !!!!!!!!!!!!!!!!!!!!!!LoadPageFault!!!!!!!!!!!!!!!!!!!!!!
[kernel] PAGE FAULT: addr:271704064 vpn:66334
[kernel] PAGE FAULT: Frame enough, allocating ppn:526102 frame.
[kernel] PAGE FAULT: Swapping in vpn:66334 ppn:526102 frame.
[kernel] PAGE FAULT: Swapping out ppn:526103 frame.
[kernel] PAGE FAULT: Mapping vpn:66334 to ppn:526102.
[kernel] !!!!!!!!!!!!!!!!!!!!!!LoadPageFault!!!!!!!!!!!!!!!!!!!!!!
[kernel] PAGE FAULT: addr:271708160 vpn:66335
[kernel] PAGE FAULT: Frame enough, allocating ppn:526103 frame.
[kernel] PAGE FAULT: Swapping in vpn:66335 ppn:526103 frame.
[kernel] PAGE FAULT: Swapping out ppn:526104 frame.
[kernel] PAGE FAULT: Mapping vpn:66335 to ppn:526103.
Test 04_1 OK!
```

图 4.2　页面置换机制简单测试结果

结果说明实现的页面置换机制基本正确。

4.2 测试用例

综合测试共七个测例，分别对每个算法进行测试，目的为检验算法正确性以及测试各个算法的效率。

七个测试用例如下：

测例一，基础测例，依顺序访问申请的内存地址并验证读写的正确性，其申请的页面（100 个）不超过用户可用的物理页面，也即所有页面可同时存放在内存中。

测例二，和测例一基本相同，但申请的物理页面若再算上申请页表用去的页面，总页面数超过了用户可用的物理页面，测试各个算法是否能正常运行并且通过测试。

测例三，使用了伪随机函数 pub fn random(m: isize, last: isize)，生成 [-m, m) 区间内的整数，last 为上一个随机数。此测例与测例一相似，大致依顺序访存，但每次访存时加上一个 [-4096, 4096) 的随机数。

测例四，使用的伪随机函数 pub fn random(m: isize, last: isize) 与测例三稍有不同，生成的是 [0, m) 区间的整数，测例进行了 4096 次整个申请内存上的随机访存并检验正确性。

测例五，与测例四相似，但将 4096 次访存分成 64 轮，将申请的内存也平均分成 64 块，每轮访问的内存为申请内存的其中一块，在这一块中进行随机访存，以模拟计算机程序的局部性，但总的访存次数不变。在测试结果中可以看见缺页异常的次数相较于测例四有显著下降。

测例六，fork 了三个子进程，每个子进程都进行测例五的测试。

测例七，fork 了三个子进程，分别进行测例二、四、五的测试。

4.3 测试结果与分析

五个算法在这七个测例上的测试结果如表 4.1所示，测试结果为缺页次数。

其中，缺页率算法后括号内的数字为参数 PFF_T 的值，工作集算法后括号内的数字为参数 WORKINGSET_DELTA_NUM 的值，测试这两种全局页面置换算法在不同参数下的结果。

可以从非常多的角度去分析这个表格：

表 4.1　页面置换算法测试结果

算法	测例一	测例二	测例三	测例四	测例五	测例六	测例七
FIFO	100	800	100	1884	2684	5084	3272
Clock	100	800	100	638	800	2400	3214
改进 Clock	100	785	100	633	781	2404	3261
缺页率 (100000)	200	800	200	7614	975	7301	10418
缺页率 (200000)	100	800	200	5999	818	3019	失败
工作集 (5)	200	800	200	7816	800	5562	11358
工作集 (20)	100	800	100	6934	800	3589	9789

- 首先，测例一、二由于访存的顺序非常规律所以得到的缺页次数基本固定且可人为计算，测例三虽然加了一个随机抖动，但整体还是从左向右的顺序访问。
- 对于完全随机的测例四，FIFO 和全局算法的表现都并不好，而两个 Clock 算法表现较好，甚至比它们在顺序访存上的表现更好，因为 FIFO 只有在用户访存非常规律时才具有较好的性能，比如线性访存，而在其他更一般的情况下效率不高，而两个全局算法对完全随机的访存也没有好的效果，需要程序的一些局部性。
- 比较完全随机的测例四和有一定局部性的随机测例五，局部算法缺页次数增加，而全局算法缺页次数减少，说明全局算法对程序局部性更敏感也更需求。
- 对于缺页率算法而言，缺页时间间隔 PFF_T 设置越大则运行时工作集越大，发生缺页的次数会更少，但有失败的风险（测例七），由于设置间隔过大页面释放不及时，虽然常驻集很大但可能超过内核可以使用的物理页面数量。当然可以进行补救，在算法中缺页时物理页面不足时释放上次缺页异常和这次之间不活跃的页面即可，这里只是单纯测试算法所以没有进行补救。
- 对于工作集算法而言，后面的数字可以代表工作集大小，工作集越大发生缺页的次数越少。
- 对比测例六和测例七：FIFO 缺页次数降低，因为其在测例五上表现最不好，而测例七中有两个子进程使用了测例二和测例四替换测例五，整体表现提升；Clock 算法和改进 Clock 算法缺页次数增加，因为局部的 Clock 算法没有考虑不同进程间的访存差异，访存规律一致时表现非常好，而访存不一

致时表现下降；全局算法表现都有下降，因为测例七中将有局部性的随机测例五换成了全随机的测例四，导致整体缺页次数增加。

- 在测例四中，全局页面置换算法的缺页次数比几种局部算法效果更差，因为系统没有对每个进程的物理页面数进行控制，而只对整体总共的页面数进行了限制，所以全局算法需要对所有进程的页面进行考虑，而局部算法依然能够基本保持它只在一个进程下的性能，若对每个进程分别进行页面限制，则局部算法会发生更多的缺页。

总的来说，对于局部页面置换算法和全局页面置换算法的测试结果都比较符合预期，并且确实能够体现不同算法之间的差异，可以看到各个算法对于不同访存模式的程序反应的结果不同。

第 5 章　多核下的同步互斥机制

如今的现代处理器基本都是支持多核的，多核的处理器可以在同一时间下各自执行不同的指令，天生支持指令的并行，可以大大提高程序的性能，所以现代的操作系统自然也需要支持多核。在教学方面，了解操作系统应该如何支持多核是有必要的，也是一种趋势。

本章将介绍如何在 rCore-Tutorial-v3 上设计支持多核的机制，这需要解决内核态同步互斥的一系列问题。

5.1　同步互斥机制的实现

想要操作系统支持多核并成功运行，首先要解决多个处理器竞争同一资源的问题。在 rCore-Tutorial-v3 的单核版本上，内核使用实现了 Sync Trait 的 UPSafeCell 类提供线程安全的保障，利用 core::cell::RefCell 提供的内部可变性对资源进行修改和管理，然而这只是提供了用户程序多线程的安全性，内核态的同步互斥问题并没有被解决。

本节将讲述如何解决多个处理器竞争同一资源的同步互斥问题，首先是使用什么工具来提供内核态多核的安全性，然后是操作系统不同模块逻辑上的修改，以支持 rCore-Tutorial-v3 在 qemu 模拟器的多核模拟环境中正常运行。

5.1.1　自旋锁

自旋锁是一种所有想要拿到它的线程都需要在循环中等待的锁，同时这些线程会在循环中不断查看是否可以获取该锁了，这就是自旋的含义。获取这种锁的线程会进行忙等待，因为这些线程虽然一直保持活跃的状态但并没有执行任何实际的任务，只是单纯地在等待。获得一个自旋锁的线程通常会一直持有该锁，直到它在代码中被显式地释放。

Rust 的 std 库提供了直接可用的自旋锁，但 rCore-Tutorial-v3 不依赖于 std 库，我们无法直接使用，但好在有一个叫 spin 的 crate 可以供我们使用。这个 crate 在 std::sync 和 std::lazy 中提供了基于自旋的原语版本，因为同步是通过自旋完成的，所以这些原语适用于 no_std 环境。

本文的实现中主要使用 spin crate 中定义的 spin::mutex::Mutex 和 MutexGuard，Mutex 是一种基于自旋的锁，提供对数据的互斥访问。将内核在单核版本中使用

的 UPSafeCell 替换为 Mutex 给需要支持同步互斥访问的资源加上自旋锁，可以让内核态的多核处理器的每个核支持对数据和资源的互斥访问。

5.1.2 多核的启动

可以支持内核态多处理器对资源的互斥访问后还不够，需要理清楚多核下操作系统每个核应该做什么，一个直接的问题是如何启动支持多核的操作系统：由哪个核来启动？启动的核需要做什么？其他核做的事有什么不同？

先介绍 rCore-Tutorial-v3 是如何启动操作系统的。在 qemu 中，系统启动后，PC 被初始化为 0x1000，经过几行汇编代码后会跳转到 0x80000000 地址，这里是 RustSBI 的代码，这意味着我们即将把控制权转交给 RustSBI，再经过一些过程后 PC 会跳转到 0x80200000 地址开始执行内核的初始化代码。所以我们使用自己写的连接脚本 linker.ld 进行链接：

```
OUTPUT_ARCH(riscv)
ENTRY(_start)
BASE_ADDRESS = 0x80200000;

SECTIONS
{
    . = BASE_ADDRESS;
    skernel = .;

    stext = .;
    .text : {
        *(.text.entry)
        . = ALIGN(4K);
        strampoline = .;
        *(.text.trampoline);
        . = ALIGN(4K);
        *(.text .text.*)
    }

    ...

    ekernel = .;
    PROVIDE(end = .);
}
```

第一行设置目标平台为 riscv，第二行设置整个程序入口点为 _start，它在下面的 entry.asm 中被定义为一个全局符号。第三行定义了常量 BASE_ADDRESS 为 0x80200000，也就是内核的初始化代码放置的位置。后面多行为内核的内存布局。

多核情况下，我们要考虑每个核不同的情况，进入初始化代码时，a0 寄存器保存了核的编号（hartid），我们可以根据这个规定每个核的栈顶位置。入口初始化代码 entry.asm 如下：

```
    .section .text.entry
    .global _start
_start:
    # a0 == hartid
    # pc == 0x80200000
    mv tp, a0

    add t0, a0, 1
    slli t0, t0, 18
    la sp, boot_stack
    add sp, sp, t0

    tail start_kernel

    .section .bss.stack
    .global boot_stack
    .global boot_stack_top
boot_stack:
    .space 256 * 1024 * 4    # 256 K per core * 4
boot_stack_top:
```

进入初始化代码后，首先将当前核的编号存入 tp（thread pointer）寄存器中，之后可以通过这个寄存器查询当前执行代码的是哪个核。然后将内核的栈空间 boot_stack 分为四部分，每个部分大小为 256K，将每个核的 sp（stack pointer）寄存器的值修改为栈顶地址。最后跳转到 start_kernel 函数的地址，该函数是现在 main.rs 中，这时内核正式开始运行，需要做一些初始化工作。

比较优雅的做法是，哪个核先执行到 start_kernel 函数，哪个核就进行初始化，但一旦有核开始初始化后，其余的核就不能再进行初始化工作了，并且要等待那个核完成初始化后，才能进行下一步操作。

core::sync::atomic 模块提供线程之间的原始共享内存通信，并且是其他并发类型的构建块。该模块定义了一些基本类型的原子版本，包括 AtomicBool、AtomicIsize、AtomicUsize、AtomicI8、AtomicU16 等。原子（Atomic）系列类型实现了可以在线程之间同步更新的一系列操作。原子类型可以存储在静态变量中，使用诸如 AtomicBool::new 之类的常量初始化器进行初始化。原子静态变量通常用于 lazy 全局初始化。这些在下面的实现中都会用到。

本文使用 AtomicBool 和 AtomicUsize 定义了如下三个原子变量，用于进行上

述初始化操作。

```
static GLOBAL_INIT_STARTED: AtomicBool = AtomicBool::new(false
  );
static GLOBAL_INIT_FINISHED: AtomicBool = AtomicBool::new(
  false);
lazy_static::lazy_static! {
    static ref BOOTED_CPU_NUM: AtomicUsize = AtomicUsize::new
  (0);
}
```

其中，GLOBAL_INIT_STARTED 表示是否已经有核在进行全局初始化，GLOBAL_INIT_FINISHED 全局初始化是否已结束。全局初始化结束后，每个核需要进行自己单独的初始化，BOOTED_CPU_NUM 表示已经完成单独初始化的核的个数。

以下函数用来进行全局初始化和每个核单独初始化：

```
fn can_do_global_init() -> bool {
    GLOBAL_INIT_STARTED.compare_exchange(false, true, Ordering
  ::Acquire, Ordering::Relaxed).is_ok()
}

fn mark_global_init_finished() {
    GLOBAL_INIT_FINISHED.compare_exchange(false, true,
  Ordering::Release, Ordering::Relaxed).unwrap();
}

fn wait_global_init_finished() {
    while GLOBAL_INIT_FINISHED.load(Ordering::Acquire) ==
  false {
        spin_loop();
    }
}

fn mark_bootstrap_finish() {
    BOOTED_CPU_NUM.fetch_add(1, Ordering::Release);
}

fn wait_all_cpu_started() {
    while BOOTED_CPU_NUM.load(Ordering::Acquire) < config::
  CPU_NUM {
        spin_loop();
    }
}
```

can_do_global_init 方法返回值表示是否还没有核进行全局初始化，如

是则返回 true，其中使用了 AtomicBool 的 compare_exchange 原子方法，若 GLOBAL_INIT_STARTED 的值为 false 则将其改为 true 后返回 true，否则不对其进行操作并返回 false，满足我们的需求，所有核中有且仅有一个核能进入全局初始化。

mark_global_init_finished 方法用来标记那些全局只执行一次的启动步骤已完成，内核必须由通过了 can_do_global_init 方法的核先启动并执行这些全局只需要一次的操作，然后其他的核才能启动。

wait_global_init_finished 方法用来等待全局初始化完成，当全局初始化正在进行时其他核进行 spin_loop。

mark_bootstrap_finish 方法用来确认、标记当前核已启动 (进行全局初始化的启动核也需要调用)，方法为给 BOOTED_CPU_NUM 用原子操作加上一。

wait_all_cpu_started 方法用来等待所有核启动完成，当原子变量 BOOTED_CPU_NUM 小于 CPU 总数时，完成了启动的核进行 spin_loop，当其等于 CPU 总数时所有核进行下一步操作。

启动函数 start_kernel 使用上述方法进行全局初始化和每个核的启动：

```
let cpu_id = arch::get_cpu_id();
if can_do_global_init() {
    println!("I am the first CPU [{}].", cpu_id);
    memory::clear_bss(); // 清空 bss 段
    memory::init();
    // memory::remap_test();
    task::add_initproc();
    println!("after initproc!");
    fs::list_apps();
    mark_global_init_finished(); // 通知全局初始化已完成
}

// 等待第一个核执行完上面的全局初始化
wait_global_init_finished();
println!("I'm CPU [{}].", cpu_id);

memory::enable_kernel_page_table();
trap::init();
arch::setSUMAccessOpen();
trap::enable_timer_interrupt();
timer::set_next_trigger();

mark_bootstrap_finish();
wait_all_cpu_started();

task::run_tasks();
```

```
unreachable!();
```

可以看到，只需要执行一次的全局初始化包括：清空 bss 段，进行堆分配器和物理页面分配器的初始化，添加初始进程 initproc，全局初始化完成后调用 mark_global_init_finished 方法通知全局初始化已完成。

在全局初始化完成后，每个核进行自己的初始化，包括：启动每个核的 SV39 分页机制，设置中断异常的入口，开启时钟中断并设置下一个时钟中断。每个核的初始化完成后调用 task::run_tasks() 开始执行用户程序。

至此，支持多核的 rCore-Tutorial-v3 操作系统就可以启动了，但还有一点小问题，将在下一小节进行讲解。

5.1.3 任务的调度

在所有核都成功启动后，每个核都会开始尝试执行用户程序，所有用户程序作为任务都放在一个队列当中：

```
lazy_static! {
    pub static ref TASK_MANAGER: Mutex<TaskManager> = Mutex::
  new(TaskManager::new());
}
```

这里使用了 5.1.1 中提到的 Mutex 锁，这个队列所有核共用，每个核需要取出下一个任务时都先获取该任务队列的 Mutex 锁，然后取出队首的任务，将其标记为 Running 后通过 __switch 切换任务上下文并通过中断异常返回至该任务的上下文处。

在单核版本的 rCore-Tutorial-v3 中，task::run_tasks() 中的循环如下代码所示：

```
let mut processor = PROCESSOR.exclusive_access();
if let Some(task) = fetch_task() {
    let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
    let mut task_inner = task.inner_exclusive_access();
    let next_task_cx_ptr = &task_inner.task_cx as *const
  TaskContext;
    task_inner.task_status = TaskStatus::Running;
    drop(task_inner);
    processor.current = Some(task);
    drop(processor);
    unsafe {
        __switch(idle_task_cx_ptr, next_task_cx_ptr);
    }
} else {
```

```
    println!("no tasks available in run_tasks");
}
```

其中 fetch_task 获取 TASK_MANAGER 的下一个可执行任务。

一开始执行时，该任务的上下文为执行第一条指令的上下文，由于处理器的调度使用的是时间片轮转，每次用户态的时钟中断时处理器会挂起当前任务并从任务队列中取出下一个任务执行，这样每次执行任务时的上下文都会不同，可能在该任务的用户程序的任何位置。

在单核版本的 rCore-Tutorial-v3 中，挂起当前任务并执行下一个的方法是这样的：

```
pub fn suspend_current_and_run_next() {
    // There must be an application running.
    let task = take_current_task().unwrap();

    // ---- access current TCB exclusively
    let mut task_inner = task.inner_exclusive_access();
    let task_cx_ptr = &mut task_inner.task_cx as *mut
  TaskContext;
    // Change status to Ready
    task_inner.task_status = TaskStatus::Ready;
    drop(task_inner);
    // ---- release current TCB

    // push back to ready queue.
    add_task(task);
    // jump to scheduling cycle
    schedule(task_cx_ptr);
}
```

先获取当前任务的可变引用，将其 TCB 中的状态更新为 Ready，然后将其加入可执行的任务队列，最后从 schedule 中的返回 __switch 到 task::run_tasks() 的循环中，从之前 __switch 结束的地方继续执行。

这样存在的问题是，因为完成上下文切换的 __switch 使用汇编代码实现的，在这段代码执行的过程中，由于在之前已经将 task_inner 的锁 drop 掉了，但在汇编中还会使用该任务的上下文，所以有极小的可能其他核在这时获取到了该任务并修改了该 task 的上下文，于是内核会出现不可预测的错误。

在目前的框架下，本文不能彻底解决这个问题，但可以使发生这种情况的概率减小，只需要将 add_task 从挂起当前任务并执行下一个任务的方法中删去，减少将 task_inner 的锁 drop 掉这一时刻到 __switch 使用完任务上下文的时刻之间的

间隔。这样做之后，需要在 run_tasks() 的 __switch 后将该任务加入可执行任务的队列。

修改后的 task::run_tasks() 中的循环如下代码所示：

```
...
if let Some(task) = fetch_task() {
    ...
    unsafe {
        __switch(idle_task_cx_ptr, next_task_cx_ptr);
    }
    let mut processor = PROCESSOR[id].lock();
    if let Some(next_task) = processor.take_current() {
        let inner = next_task.inner_lock();
        if inner.task_status == TaskStatus::Ready{
            drop(inner);
            add_task(next_task);
        }
    }
    drop(processor);
}
```

在 __switch 返回循环后再将该 task 加入可执行的任务队列中。

修改后的 suspend_current_and_run_next 如下代码所示：

```
pub fn suspend_current_and_run_next() {
    ...
    task_inner.task_status = TaskStatus::Ready;
    drop(task_inner);
    schedule(task_cx_ptr);
}
```

在 drop 了 task_inner 的锁后立即返回。

5.2 测试结果

在 qemu 中设置多核数量为四核，并在 config 中设定 CPU_NUM 为 4，启动操作系统后可以成功运行，如图 5.1所示。

usertests 为 rCore-Tutorial-v3 设计的用户程序的测试用例集，里面包括了简单输出测试（hello_world）、炫酷文字输出、fork 测试、矩阵乘法计算、sleep 测试、yield 测试和栈溢出测试，本文实现的支持多核的版本可以运行并通过测试。

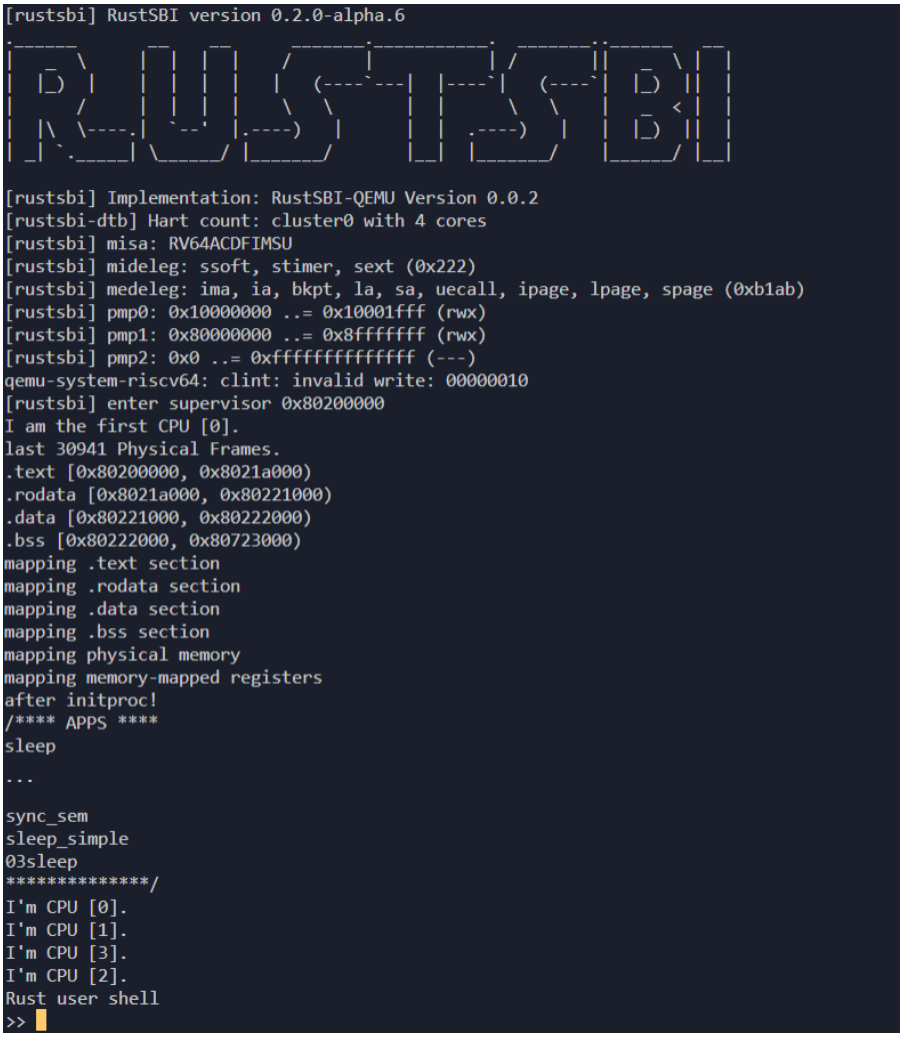

图 5.1　支持多核的 rCore-Tutorial-v3 启动结果

第 6 章　项目总结

在现在的操作系统中，拥有稳定且具有较好性能的页面置换算法和高效页面置换机制是必须且至关重要的。本文以教学为目的，在已有的 rCore-Tutorial-v3 上探索并完成了逻辑清晰、易于理解且实现简单的一套页面置换机制，在该机制上实现了三种局部页面置换算法和两种全局页面置换算法，并设计了一套完整的测试用例对这五种算法进行了测试和分析。

本文按照 rCore-Tutorial-v3 的章节结构，一步一步实现了对应章节支持多核的版本，解决了内核态同步互斥问题，并在 qemu 模拟器中成功运行并通过测试。

为了完成上述机制功能的实现，我学习了更多的 Rust、RISCV 和操作系统相关的知识，解决了遇到的许多困难，这个过程使得我对操作系统的认知更加深刻，对一个系统项目的开发有了更多的了解。

插图索引

表格索引

参考文献

[1] Waterman A, Asanovic K, Inc S. The risc-v instruction set manual volume ii: Privileged architecture[M]. CS Division, EECS Department, University of California, Berkeley, 2020: 77.

[2] Waterman A, Asanovic K, Inc S. The risc-v instruction set manual volume ii: Privileged architecture[M]. CS Division, EECS Department, University of California, Berkeley, 2020: 68.

致　　谢

衷心感谢导师陈渝副教授对本人的精心指导。他的言传身教将使我终生受益，他在我的研究过程中多次帮助我解决了遇到的困难。

感谢闭浩扬同学，他的项目给了我指导和启发，为我节省了大量的时间。

感谢我的朋友们，他们给了我一个更好的心态和美好的心情，让我能安心地学习和工作。

感谢本科期间的各位老师的认真教导，感谢四年来身边同学的陪伴和帮助。

声　　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　外文资料的书面翻译

操作系统历险记：使用 Rust 编写 RISC-V 操作系统

目录

A.1 外部中断

2019 年 11 月 18 日: 仅 Patreon
2019 年 11 月 25 日: 公开

A.1.1 视频

https://www.youtube.com/watch?v=99KMubPgDIU

A.1.2 概述

在上一章，我们讨论了 CPU 以及内核的中断。在这一章，我们将讨论中断的其中一类——外部中断，这些中断表示发生了某些外部或平台中断。例如，UART 设备可能刚刚填满了它的缓冲区。

A.1.3 平台级中断控制器 (The Platform-Level Interrupt Controller)

平台级中断控制器 (PLIC) 通过 CPU 上的一个引脚——EI (外部中断) 引脚，来路由所有信号，通过 mie 寄存器中的机器外部中断启用 (meie) 位可以启用该引脚。

每当我们看到该引脚已被触发（外部中断待处理）时，我们就可以查询 PLIC 以查看是什么原因造成的。此外，我们可以将 PLIC 配置为优先考虑中断源或完全禁用某些源，同时启用其他源。

A.1.4 PLIC 寄存器

PLIC 是一个通过 MMIO 控制的中断控制器。有几个与 PLIC 相关的寄存器：

寄存器	地址	描述
Priority	0x0c00_0000	设置特定中断源的优先级
Pending	0x0c00_1000	包含已触发的中断列表(待处理)
Enable	0x0c00_2000	启用/禁用某些中断源
Threshold	0x0c20_0000	设置中断触发的阈值
Claim(read)	0x0c20_0004	按优先级顺序返回下一个中断
Complete(write)	0x0c20_0004	完成对特定中断的处理

PLIC 通过外部中断引脚连接到 CPU。以下架构图 A.1来自SiFive's Freedom Unleashed Manual

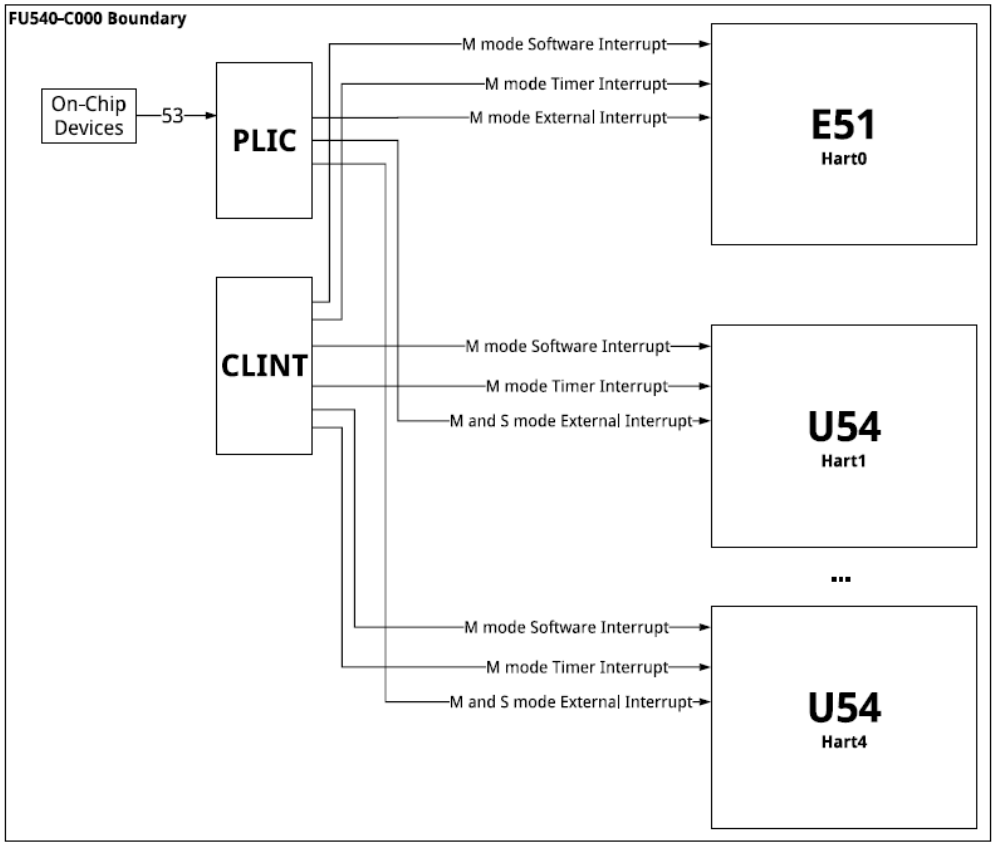

Figure 3: FU540-C000 Interrupt Architecture Block Diagram.

图 A.1 PLIC 架构图

PLIC 连接到外部设备，并通过位于 PLIC 基地址（如上表所示的寄存器）的可编程接口控制它们的中断。这意味着我们作为程序员可以控制每个中断的优先级，我们是否看到它，我们是否处理过它，等等。

上图可能有点混乱，但在 QEMU 中编程的 PLIC 要简单得多。

A.1.5 PLIC 处理

系统将通过电线将 PLIC 连接到外部设备，我们通常会在技术参考文档或类似文件中获得线号。但是对于我们来说，我在 qemu/include/hw/riscv/virt.h 中查看到 UART 连接到引脚 10，VIRTIO 设备是 1 到 8，PCI express 设备是 32 到 35。你可能在想为什么在我们的 Rust 代码中，我们只有一个中断启用寄存器——因为我们不会为了我们的目的而超过中断 10 (UART)。

A.1.6 启用中断源

现在知道了设备通过哪个中断进行连接，我们通过将 1 « id 写入中断启用寄存器来启用该中断。此示例的中断 ID 将为 UART 的 10。

```
/// 启用给定的中断 id
pub fn enable(id: u32) {
    let enables = PLIC_INT_ENABLE as *mut u32;
    let actual_id = 1 << id;
    unsafe {
        // 与 complete 和 claim 寄存器不同，plic_int_enable
        // 寄存器是一个位集(bitset)，其中 id 是位索引。
        // 该寄存器是一个 32 位寄存器，因此我们可以启用
        // 中断 31 到 1（0 硬连线到 0）。
        enables.write_volatile(enables.read_volatile() |
  actual_id);
    }
}
```

A.1.7 设置中断源优先级

现在已经启用了中断源，我们需要给它一个 0 到 7 的优先级。7 是最高优先级，0 是“scum”类（h/t Top Gear）——但是优先级 0 不能满足任何阈值，因此它基本上禁用了中断（请参阅下面的 PLIC 阈值）。我们可以通过优先级寄存器设置每个中断源的优先级。

```
/// 将给定中断的优先级设置为给定优先级。
/// 优先级必须为 [0..7]
pub fn set_priority(id: u32, prio: u8) {
    let actual_prio = prio as u32 & 7;
    let prio_reg = PLIC_PRIORITY as *mut u32;
    unsafe {
        // 中断 id 的偏移量是:
```

```
        // PLIC_PRIORITY + 4 * id
        // 由于我们在 u32 类型上使用指针运算
        // 它会自动将 id 乘以 4。
        prio_reg.add(id as usize).write_volatile(actual_prio);
    }
}
```

A.1.8 设置 PLIC 阈值

PLIC 本身有一个全局阈值，所有中断在“启用”之前都必须通过该阈值。这是通过阈值寄存器控制的，我们可以将值 0 到 7 写入其中。任何小于或等于此阈值的中断优先级都无法满足障碍并被屏蔽，这实际上意味着中断被禁用。

```
/// 设置全局阈值，阈值可以是值 [0..7]。
/// PLIC 将屏蔽等于或低于给定阈值的任何中断。
/// 这意味着阈值 7 将屏蔽所有中断，阈值 0 将允许所有中断。
pub fn set_threshold(tsh: u8) {
    // 我们使用 tsh 是因为我们使用的是 u8
    // 但我们的最大数量是 3 位 0b111。
    // 所以，我们与 7 (0b111) 求 and 来获取最后三位。
    let actual_tsh = tsh & 7;
    let tsh_reg = PLIC_THRESHOLD as *mut u32;
    unsafe {
        tsh_reg.write_volatile(actual_tsh as u32);
    }
}
```

A.1.9 处理 PLIC 中断

PLIC 将通过异步原因 (asynchronous cause) 11（机器外部中断）向我们的操作系统发出信号。当处理这个中断时，我们不会知道实际上是什么导致了中断——只知道 PLIC 导致了它。这是声明 (claim)/完成 (complete) 过程开始的地方。

A.1.10 声明一个中断

SiFive 将声明/完成过程描述如下：

10.7 中断声明过程

FU540-C000 hart 可以通过读取 claim/complete 寄存器（表 45）来执行中断声明，该寄存器返回最高优先级的挂起中断的 ID，如果没有挂起的中断，则返回零。成功的声明还会原子地清除中断源上相应的 pending 位。FU540-C000 hart 可以随时执行声明，即使其 mip（表 22）寄存器中的 MEIP 位未设置。

声明操作不受优先级阈值寄存器设置的影响。

10.8 中断完成

FU540-C000 hart 通过将其从声明中接收到的中断 ID 写入 claim/complete 寄存器（表 45）来发出它已完成执行中断处理程序的信号。PLIC 不检查完成 ID 是否与该目标的最后一个声明 ID 相同。如果完成 ID 与当前为目标启用的中断源不匹配，则 g 该完成会被静默忽略。

claim 寄存器将返回按优先级排序的下一个中断。如果寄存器返回 0，则没有挂起的中断，这不应该发生，因为我们将通过中断处理程序处理声明过程。

```
/// 获取下一个可用中断，这就是“claim声明”过程。
// plic 会自动按优先级排序，并将中断的 ID 交给我们。
// 例如，如果 UART 正在中断并且为下一个，我们将得到值 10。
pub fn next() -> Option {
    let claim_reg = PLIC_CLAIM as *const u32;
    let claim_no;
    // 声明寄存器填充了最高优先级的已启用中断。
    unsafe {
        claim_no = claim_reg.read_volatile();
    }
    if claim_no == 0 {
        // 中断 0 硬连线到 0，这告诉我们没有要声明的中断
        // 因此我们返回 None。
        None
    }
    else {
        // 如果到达这里，我们就会得到一个非 0 中断。
        Some(claim_no)
    }
}
```

然后，我们可以要求 PLIC 通过该声明寄存器向我们提供任何发起中断的编号。在 Rust 中，我通过匹配将其驱动到正确的处理程序。到目前为止，我们直接在中断上下文中处理它——这是一个坏主意，但我们还没有任何延迟任务系统。

```
// 机器外部 (来自 PLIC 的中断)
// println!("Machine external interrupt CPU#{}", hart);
// 我们将检查下一个中断。 如果中断不可用，则将给我们 None。
// 然而，这意味着我们得到了一个虚假的中断，除非
// 我们得到一个来自非 PLIC 源的中断。 这是 PLIC 将 id 0 硬连线
// 到 0 的主要原因，因此我们可以将其用作错误案例。
if let Some(interrupt) = plic::next() {
  // 如果我们到达这里，我们就会从声明寄存器中得到一个中断。
  // PLIC 将自动为下一个中断设置优先级，因此当我们从声明中获取它时，
  // 它将是优先级顺序中的下一个。
  match interrupt {
  10 => { // 中断 10 是 UART 中断。
    // 我们通常会将其设置为在中断上下文之外进行处理，
    // 但我们在这里进行测试！ 来吧！
    // 我们还没有为 my_uart 使用单例模式，但请记住，
    // 这只是简单地包装(wrap)了 0x1000_0000 (UART)。
    let mut my_uart = uart::Uart::new(0x1000_0000);
    // 如果我们到了这里，UART 最好有一些东西！ 如果不是，会发生什么？？
    if let Some(c) = my_uart.get() {
    // 如果您认识这段代码，它曾经位于 kmain() 下的 lib.rs 中。
    // 那是因为我们需要轮询 UART 数据。
    // 既然现在我们有中断了，那就来吧！
    match c {
      8 => {
      // 这是一个退格，所以我们基本上必须写一个空格，再写一个退格：
      print!("{} {}", 8 as char, 8 as char);
      },
      10 | 13 => {
      // 换行或回车
      println!();
      },
      _ => {
      print!("{}", c as char);
      },
    }
    }

  },
  // 非UART中断在这里并且什么也不做。
  _ => {
    println!("Non-UART external interrupt: {}", interrupt);
  }
  }
  // 我们已经声明了它，所以现在假设我们已经处理了它。
  // 这将重置挂起的中断并允许 UART 再次中断。 否则，UART 将“卡住”。
  plic::complete(interrupt);
}
```

A.1.11 告诉 PLIC 我们需要擦除 (wiping)

当我们声明一个中断时，我们是在告诉 PLIC 它将被处理或正在被处理。在此期间，PLIC 不会再监听来自同一设备的任何中断。这是 complete 过程的开始。当我们写入（而不是读取）声明寄存器时，我们给出一个中断的值来告诉 PLIC 我们已经完成了它给我们的中断。然后，PLIC 可以重置中断触发器并等待该设备将来再次中断。这会重置系统并让我们一遍又一遍地循环回到声明/完成。

```
// 通过 id 完成一个挂起的中断。 id 应该来自上面的 next() 函数。
pub fn complete(id: u32) {
    let complete_reg = PLIC_CLAIM as *mut u32;
    unsafe {
        // 我们实际上将一个 u32 写入整个 complete_register。
        // 这与声明寄存器是同一个寄存器，
        // 但它可以根据我们是在读还是在写来区分。
        complete_reg.write_volatile(id);
    }
}
```

就是这样。PLIC 已被编程，现在它只处理我们的 UART。请注意在 lib.rs 中我删除了 UART 轮询代码。现在有了中断，我们只能在它向我们发出信号时处理 UART。由于我们使用等待中断 (wfi) 将 HART 置于等待循环中，我们可以节省电力和 CPU 周期。

让我们将它添加到我们的 kmain 函数中，看看会发生什么！

```
// 让我们通过 PLIC 设置中断系统。
// 我们必须将阈值设置为不会屏蔽所有中断的值。
println!("Setting up interrupts and PLIC...");
// 我们降低了阈值墙，这样我们的中断就可以跃过它。
plic::set_threshold(0);
// VIRTIO = [1..8]
// UART0 = 10
// PCIE = [32..35]
// 启用 UART 中断。
plic::enable(10);
plic::set_priority(10, 1);
println!("UART interrupts have been enabled and are awaiting
  your command");
```

当我们运行 (make run) 这段代码时，我们得到以下信息 A.2。我输入了底线 (bottom line)，但它表明我们现在正在通过 PLIC 处理中断！

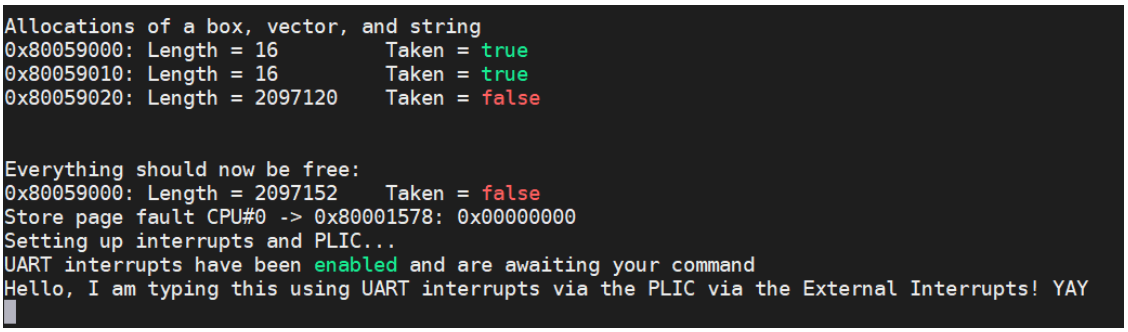

图 A.2　PLIC 运行图

A.2　进程的内存

2019 年 12 月 08 日: 仅 Patreon

2019 年 12 月 23 日: 公开

A.2.1　概述

进程是操作系统的重点。我们想开始“做点事”，我们将把它融入一个进程并让它能够运行。我们将在未来随着向进程添加功能来更新进程结构体，现在我们需要一个程序计数器（正在执行的指令，pc）和一个用于本地内存的栈。

我们不会为进程创建标准库。在本章中，我们将编写内核函数并将它们包装到一个进程中。当我们开始创建我们的用户进程时，我们需要从块设备中读取并开始执行指令。这是一种相当不错的方式，因为我们将需要系统调用等等。

A.2.2　进程结构体

每个进程都必须包含一些信息，以便我们可以运行它。我们需要知道一些信息，例如所有的寄存器、MMU 表和进程的状态。

这个进程结构体尚未完成，我曾想过现在就完成，但我认为逐步完成它会帮助我们理解需要什么以及为什么该部分需要存储在进程结构体中。

```
pub enum ProcessState {
  Running,
  Sleeping,
  Waiting,
  Dead,
}

#[repr(C)]
pub struct Process {
  frame:            TrapFrame,
```

```
  stack:           *mut u8,
  program_counter: usize,
  pid:             u16,
  root:            *mut Table,
  state:           ProcessState,
}
```

该进程包含一个栈 (stack)，我们将把它提供给 sp（栈指针）寄存器，但是这个 stack 显示的是栈的顶部，所以我们需要添加大小 (size)。请记住，一个栈是从高内存增长到低内存的（用减法分配并用加法释放）。

程序计数器 (program_counter) 将包含下一个要执行的指令的地址，我们可以通过 mepc 寄存器（机器异常程序计数器）得到它。当我们使用上下文切换定时器或非法指令亦或是作为对 UART 输入的响应来中断我们的进程时，CPU 会将即将执行的指令放入 mepc，我们可以通过返回这里重新开始我们的过程并再次开始执行指令。

A.2.3 陷入帧 (Trap Frame)

我们目前使用的是单处理器系统，我们以后会把它升级成一个多 hart 系统，但对现在来说就这样会更有意义。陷入帧允许我们在通过内核处理进程时冻结进程，这在来回切换时是必须的，你能想象你刚刚设置了寄存器它就被内核修改了吗？

进程由其上下文定义。TrapFrame 显示了上下文在 CPU 上的样子：(1) 通用寄存器（有 32 个），(2) 浮点寄存器（也有 32 个），(3) MMU，(4) 一个栈用来处理这个进程的中断上下文。我还存储了 hartid，这样我们就不会同时在两个 hart 上运行相同的进程。

```
#[repr(C)]
#[derive(Clone, Copy)]
pub struct TrapFrame {
  pub regs:       [usize; 32], // 0 - 255
  pub fregs:      [usize; 32], // 256 - 511
  pub satp:       usize,       // 512 - 519
  pub trap_stack: *mut u8,     // 520
  pub hartid:     usize,       // 528
}
```

A.2.4 创建一个进程

我们需要为进程的栈和进程结构体本身分配内存。我们将使用 MMU，因此我们可以为所有进程提供一个已知的起始地址，我选择了 0x2000_0000。当我们创建外部进程并编译它们时，我们需要知道起始内存地址，由于这是虚拟内存，它本质上可以是我们想要的任何值。

```
impl Process {
  pub fn new_default(func: fn()) -> Self {
    let func_addr = func as usize;
    // 当我们开始进行多 hart 处理时，
    // 我们会将下面的 NEXT_PID 转换为一个原子(atomic)增量
    // 现在我们需要一个进程，让它可以工作，之后再改进它！
    let mut ret_proc =
    Process { frame:           TrapFrame::zero(),
          stack:           alloc(STACK_PAGES),
          program_counter: PROCESS_STARTING_ADDR,
          pid:             unsafe { NEXT_PID },
          root:            zalloc(1) as *mut Table,
          state:           ProcessState::Waiting,
          data:            ProcessData::zero(), };
    unsafe {
    NEXT_PID += 1;
    }
    // 现在我们将栈指针移动到分配的底部。
    // 规范显示寄存器 x2 (2) 是栈指针。
    // 我们可以使用 ret_proc.stack.add，
    // 但这是一个不安全的函数，需要一个不安全块。
    // 所以，先转换成usize再加上PAGE_SIZE比较好。
    // 我们还需要设置栈调整，使其位于内存的底部并且远离堆分配。
    ret_proc.frame.regs[2] = STACK_ADDR + PAGE_SIZE *
  STACK_PAGES;
    // 在 MMU 上映射栈
    let pt;
    unsafe {
    pt = &mut *ret_proc.root;
    }
    let saddr = ret_proc.stack as usize;
    // 我们需要将栈映射到用户进程的虚拟内存。
    // 这有点麻烦，因为我们还需要映射函数代码。
    for i in 0..STACK_PAGES {
    let addr = i * PAGE_SIZE;
    map(
      pt,
      STACK_ADDR + addr,
      saddr + addr,
      EntryBits::UserReadWrite.val(),
      0,
```

```
    );
    }
    // 在 MMU 上映射程序计数器
    map(
      pt,
      PROCESS_STARTING_ADDR,
      func_addr,
      EntryBits::UserReadExecute.val(),
      0,
    );
    map(
      pt,
      PROCESS_STARTING_ADDR + 0x1001,
      func_addr + 0x1001,
      EntryBits::UserReadExecute.val(),
      0,
    );
    ret_proc
  }
}

impl Drop for Process {
  // 由于我们将 Process 的所有权存储在链表中,
  // 我们可以使其在被删除时自动释放。
  fn drop(&mut self) {
    // 我们将栈分配为一个页面。
    dealloc(self.stack);
    // 这是不安全的，但它处于丢弃(drop)阶段，所以我们不会再使用它。
    unsafe {
    // 请记住，unmap会取消映射除根以外的所有级别的页表。
    // 它还释放与表关联的内存。
    unmap(&mut *self.root);
    }
    dealloc(self.root as *mut u8);
  }
}
```

现在 Process 结构的实现相当简单，Rust 要求我们对所有字段都有一些值，所以我们创建这些辅助函数来做到这一点。

当我们创建一个进程时，我们并没有真正“创建”它，相反，我们分配内存来保存进程的元数据。目前，所有进程代码都存储为一个 Rust 函数，之后我们将从块设备加载二进制指令并以这种方式执行。请注意，我们需要做的就是创建一个栈。我为一个栈分配了 2 个页面，给了栈 4096 * 2 = 8192 字节的内存，这很小，但对于我们正在做的事情来说应该绰绰有余。

A.2.5 CPU 级别的进程

每当我们遇到 trap 时，都会处理 CPU 级别的进程。我们将使用 CLINT 计时器作为我们的上下文切换计时器。现在我已经为每个进程分配了 1 整秒的时间，这对于正常操作来说太慢了，但它使调试变得更容易，因为我们可以看到一个进程执行的步骤。

每次 CLINT 计时器命中时，我们都将存储当前上下文，跳转到 Rust（通过 trap.rs 中的 m_trap），然后处理需要做的事情。我选择使用机器模式来处理中断，因为我们不必担心切换 MMU（它在机器模式下关闭）或遇到递归错误。

CLINT 定时器通过 MMIO 控制如下：

```
unsafe {
  let mtimecmp = 0x0200_4000 as *mut u64;
  let mtime = 0x0200_bff8 as *const u64;
  // QEMU 给出的频率是 10_000_000 Hz，
  // 因此这会将下一个中断设置为从现在开始一秒后触发。
  mtimecmp.write_volatile(mtime.read_volatile() + 10_000_000);
}
```

mtimecmp 是存储未来时间的寄存器，当 mtime 达到这个值时，它会以“机器定时器中断”的原因中断 CPU，我们可以使用这个周期性计时器来中断我们的进程并切换到另一个。这是一种称为“时间切片”的技术，我们将 CPU 时间的切片分配给每个进程。定时器中断的速度越快，我们在给定时间内可以处理的进程就越多，但是每个进程只能执行一小部分指令。此外，每个中断都会执行上下文切换（m_trap_handler）代码。

定时器频率有点像一门艺术，Linux 在将其设定为 1,000Hz（每秒一千次中断）时深有体会，这意味着每个进程能够运行 1/1000 秒。诚然，以当今处理器的速度，这可以执行大量的指令，但在过去，这是有争议的。

A.2.6 带陷入帧 (Trap Frame) 的陷入处理程序 (Trap Handler)

我们的陷入处理程序必须能够处理不同的陷入帧，因为当我们需要命中陷入处理程序时，任何进程（甚至内核）都可能在运行。

```
m_trap_vector:
# 这里所有的寄存器都是易失的，我们需要在做任何事情之前保存它们。
csrrw t6, mscratch, t6
# csrrw 将自动将 t6 到 mscratch 中并将 mscratch 的旧值交换为 t6。
# 这很好，因为我们只是切换了值并且没有破坏任何东西——所有的都是原子的！
```

```
# 在 cpu.rs 中我们有一个结构:
#  32 gp regs   0
#  32 fp regs   256
#  SATP register  512
#  Trap stack       520
#  CPU HARTID   528
# 我们使用 t6 作为临时寄存器，因为它是最底层的寄存器 (x31)
.set  i, 1
.rept 30
  save_gp %i
  .set  i, i+1
.endr

# 保存实际的 t6 寄存器，我们将其交换到 mscratch
mv    t5, t6
csrr  t6, mscratch
save_gp 31, t5

# 将内核陷入帧(kernel trap frame)恢复到 mscratch
csrw  mscratch, t5

# 准备好进入 Rust (trap.rs)
# 我们不想写入用户栈或任何在这有关的东西。
csrr  a0, mepc
csrr  a1, mtval
csrr  a2, mcause
csrr  a3, mhartid
csrr  a4, mstatus
mv    a5, t5
ld    sp, 520(a5)
call  m_trap

# 当我们到达这里时，我们已经从 m_trap 返回，恢复寄存器并返回。
# m_trap 将通过 a0 返回返回地址。

csrw  mepc, a0

# 现在将陷入帧加载回 t6
csrr  t6, mscratch

# 恢复所有通用寄存器
.set  i, 1
.rept 31
  load_gp %i
  .set  i, i+1
.endr

# 由于我们从 i = 1 开始运行此循环 31 次，
# 因此最后一个循环将 t6 加载回其原始值。

mret
```

我们必须手动操作 TrapFrame 字段，因此了解偏移量很重要。每当我们更改 TrapFrame 结构时，我们都需要确保我们的汇编代码反映了这种更改。

在汇编代码中，我们可以看到我们做的第一件事就是冻结当前正在运行的进程。我们使用 mscratch 寄存器来保存当前正在执行的进程（或内核——它使用 KERNEL_TRAP_FRAME）的 TrapFrame，csrrw 指令将自动存储 t6 寄存器的值并将旧值（陷入帧的内存地址）返回到 t6。请记住，我们必须保持所有寄存器的原始值，这是一个很好的方法！

出于方便，我们使用 t6 寄存器。它是 32 号寄存器（索引 31），所以它是我们循环保存或恢复寄存器时的最后一个寄存器。

如上所示，我使用 GNU 的.altmacro 的宏循环来保存和恢复寄存器，很多时候您会看到所有 32 个寄存器都被保存或恢复，但我这样做显著地简化了代码。

```
csrr  a0, mepc
csrr  a1, mtval
csrr  a2, mcause
csrr  a3, mhartid
csrr  a4, mstatus
mv    a5, t5
ld    sp, 520(a5)
call  m_trap
```

这部分陷入处理程序用于将参数转发给 rust 函数 m_tra，如下所示：

```
extern "C" fn m_trap(epc: usize,
    tval: usize,
    cause: usize,
    hart: usize,
    status: usize,
    frame: *mut TrapFrame) -> usize
```

按照 ABI 约定，a0 是 epc，a1 是 tval，a2 是 cause，a3 是 hart，a4 是 status，a5 是指向陷入帧的指针。您还会注意到我们返回了一个 usize，这用于返回要执行的下一条指令的内存地址。请注意，调用 m_trap 之后的下一条指令是将 a0 寄存器加载到 mepc 中。同样按照 ABI 约定，Rust 的返回值存储在 a0 中。

A.2.7 第一个进程——Init

我们的调度算法（稍后）将始终需要至少一个进程，就像 Linux 一样，我们将把这个进程称为 init。

本章的重点是展示一个进程在抽象视图中的样子，我将通过添加调度程序并显示实际运行的进程来在本章基础上进一步展示，这将是我们操作系统的基础。想要让进程实际执行操作，我们需要做的另一件事是实现系统调用，因此您可以期待它！

A.3 系统调用

2020 年 1 月 23 日: 仅 Patreon

2020 年 1 月 29 日: 公开

A.3.1 视频

https://www.youtube.com/watch?v=6GW_jgkdGPw

A.3.2 概述

系统调用是非特权用户应用程序向内核请求服务的一种方式。在 RISC-V 架构中，我们使用 ecall 指令调用，这将导致 CPU 停止它正在做的事情，提升特权模式，然后跳转到存储在 mtvec（机器陷入向量）寄存器中的任何函数处理程序，请记住这是处理所有陷入的“漏斗”，包括我们的系统调用。

我们必须设置处理系统调用的约定，可以使用已经存在的约定，这样可以与库进行交互，例如 newlib。但是，让我们把它变成我们的！我们可以规定系统调用号是多少，以及当我们执行系统调用时它们将在哪里。

A.3.3 系统调用程序

我们通常只需要在处于较低权限模式时执行系统调用，如果我们在内核中，我们已经可以访问大多数特权系统，这使我们实际上不需要进入系统调用。

我们的系统调用是通过同步陷入 8 到达的，这是用户模式 ecall 的原因 (cause)。因此在我们的 8 处理程序中，我们将数据转发到我们的系统调用处理程序。我们将完全使用 Rust，我们还需要能够操作程序计数器。想想看，我们的 exit 系统调用必须能够移动到另一个进程，所以我们通过 mepc（机器异常程序计数器）寄存器来操作它。

A.3.4 Rust 系统调用

与往常一样，请确保使用以下内容导入系统调用代码。

```
pub mod syscall;
```

这将进入您的 lib.rs 文件。

A.3.5 顺序与编号

一些库已经有它们希望您的系统调用保持的顺序，但是我们将为我们的简单应用程序创建自己的“C 库”，因此只要我们保持一致，就可以继续下去。

想想我们如何才能做到这一点。每当我们执行 ecall 指令时，CPU 都会提升权限并跳转到陷阱向量，我们如何发送数据呢？ARM 架构允许您将数字编码到他们的 svc（superviser 调用）指令中，然而许多操作系统放弃了这种实现，那么我们该怎么做呢？

答案是：寄存器。我们在 RISC-V 架构中有大量的寄存器，所以我们几乎不像在 x86 时代那样受到限制。对于我们的系统调用约定，我们将系统调用的编号放入第一个参数寄存器 a0，后续参数将进入 a1、a2、a3、...、a7，然后我们将使用相同的 a0 寄存器进行返回。

这与常规函数的调用约定相同，因此它将与我们已经知道的内容很好地交互。在 RISC-V 中，我们可以使用伪指令 call 来进行正常的函数调用，或者使用 ecall 来进行 superviser 调用。一致性是关键。Consistency is cey, or Konsistency is Key. Hmm.. Consistency is key isn’t consistently sounding the ’k’ :(.

A.3.6 实现系统调用

我们将同步原因 (cause) 8 重定向到我们的系统调用 Rust 代码。

```
8 => {
  // 来自用户模式的环境（系统）调用
  println!("E-call from User mode! CPU#{} -> 0x{:08x}", hart,
  epc);
  return_pc = do_syscall(return_pc, frame);
},
```

大多数操作系统使用函数指针构建一个表，但我在这里使用 Rust 的 match 语句。我没有进行任何性能计算，但我不认为一个比另一个有明显的优势。再说一次，不要引用我的话，我还没有实际测试过。

如您所见，我们收到一个新的程序计数器，它是我们返回时要执行的指令的地址，系统调用函数必须至少将其加 4，因为 ecall 指令实际上是导致同步中断的原因，如果我们不移动程序计数器，我们就会一遍又一遍地执行 ecall 指令。幸运的是，与 x86 不同的是，除了 16 位压缩指令外，所有指令都是 32 位的，但 ecall 始终是 32 位的，因为它没有压缩形式。

A.3.7 神说：要有系统调用！

让我们看一下 Rust 中的代码。

```
pub fn do_syscall(mepc: usize, frame: *mut TrapFrame) -> usize
   {
    let syscall_number;
    unsafe {
        // A0 是 X10，所以它的寄存器号是 10。
        syscall_number = (*frame).regs[10];
    }
    match syscall_number {
        0 => {
            // Exit
            println!("You called the exit system call!");
            mepc + 4
        },
      _ => {
            print!("Unknown syscall number {}", syscall_number
  );
            mepc + 4
        }
    }
}
```

我们首先需要的是 A0 的值，它是系统调用号。由于它在陷入处理程序阶段存储在上下文中，我们可以直接从内存中检索它。我们必须把它放在一个 unsafe 上下文中，因为我们正在取消引用一个原始 (raw) 指针，它可能是也可能不是准确的内存地址，由于 Rust 不能保证它是，我们需要把它放在一个不安全的块中。

这对非 Rustaceans 来说可能很有趣。

```
let syscall_number;
unsafe {
    // A0 是 X10，所以它的寄存器号是 10。
    syscall_number = (*frame).regs[10];
}
```

我正在创建一个名为 syscall_number 的变量，但由于在进入不安全块之前我无法获得它的值，所以它只是一个占位符。事实上，在我们给它一个值之前，Rust 都不会给它一个类型。你会注意到我没有对变量类型施加任何限制，所以我让 Rust 来决定。

我为什么这样做？unsafe 块创建了一个新块，因此它创建了一个新范围 (scope)，但是我希望 syscall_number 在不安全上下文之外包含一个不可变值，这就是为什么我决定这样做。从技术上讲，我可以使用 let syscall_number: u64; 来约束数据类型，但这不是必需的，因为只要我们将变量设置为等于某个值，Rust 就会评估数据类型。

A.3.8 现在我们干嘛？

我们将编写调度程序，以便我们的系统调用实际上可以做一些事情——是的，以技术上最准确的方式做事情！例如，我们可能需要将进程推迟到一定时间后（有点像 sleep() 的工作方式），或者我们需要关闭进程（很像 exit() 的工作方式），写点东西到控制台怎么样——是的，我们也需要那个。

因此，接下来我们将添加进程和必要的系统调用，我们没有做出任何未来的预测，这可能很危险，但我们正在实现我们的操作系统，因为我们发现了一个不可行的解决方案。希望这将使您了解操作系统怎样实现为所有应用程序的一切！

A.4 启动一个进程

2020 年 3 月 10 日: 仅 Patreon

2020 年 3 月 16 日: 公开

A.4.1 视频和参考资料

我在我的大学里教过操作系统，所以我将在此处链接我在该课程中关于进程的笔记。

https://www.youtube.com/watch?v=eB3dkJ2tBK8

OS Course Notes: Processes

上面的笔记是对进程作为一个概念的一般概述，我们在这里构建的操作系统可能会不太一样，大部分是因为它是用 Rust 编写的——在这里插入笑话。

A.4.2 概述

启动一个进程是我们一直在等待的，操作系统的工作本质上是支持正在运行的进程，在这篇文章中，我们将从操作系统的角度以及 CPU 的角度来看一个进程。

我们在上一章中查看了进程内存，但其中一些已被修改，以便我们拥有一个常驻内存空间（在堆上）。此外，我将向您展示如何从内核模式进入用户模式，现在我们已经删除了主管 (supervisor) 模式，但是当回顾系统调用以支持进程时，我们会修复它。

A.4.3 进程结构体

进程结构大致相同，但在 CPU 方面，我们只关心 TrapFrame 结构。

```
#[repr(C)]
#[derive(Clone, Copy)]
pub struct TrapFrame {
  pub regs:       [usize; 32], // 0 - 255
  pub fregs:      [usize; 32], // 256 - 511
  pub satp:       usize,       // 512 - 519
  pub trap_stack: *mut u8,     // 520
  pub hartid:     usize,       // 528
}
```

我们不会使用所有这些字段，而现在我们只关心寄存器上下文（pub regs）。当我们捕获一个陷入时，我们会将当前在 CPU 上执行的进程存储到 regs 陷入帧中，因此我们在处理陷入时保留该过程并冻结它。

```
csrr  a0, mepc
csrr  a1, mtval
csrr  a2, mcause
csrr  a3, mhartid
csrr  a4, mstatus
csrr  a5, mscratch
la    t0, KERNEL_STACK_END
ld    sp, 0(t0)
call  m_trap
```

在陷入中，在我们保存了上下文之后，我们开始向 Rust 陷入处理程序 m_trap 提供信息，这些参数必须与 Rust 中的顺序匹配。最后，请注意我们将 KERNEL_STACK_END 放入堆栈指针。当我们保存它们时，实际上没有任何寄存器发生变化（除了 a0-a5、50 和现在的 sp），但是当我们跳转到 Rust 时，我们需要

一个内核栈。

A.4.4 调度

我添加了一个非常简单的调度程序，它只是轮换进程列表，然后检查最前面的进程。目前还没有办法改变进程状态，但是每当我们找到一个正在运行的进程时，我们就会抓取它的数据，然后将它放在 CPU 上。

```
pub fn schedule() ->  (usize, usize, usize) {
  unsafe {
    if let Some(mut pl) = PROCESS_LIST.take() {
    pl.rotate_left(1);
    let mut frame_addr: usize = 0;
    let mut mepc: usize = 0;
    let mut satp: usize = 0;
    let mut pid: usize = 0;
    if let Some(prc) = pl.front() {
      match prc.get_state() {
      ProcessState::Running => {
        frame_addr =
        prc.get_frame_address();
        mepc = prc.get_program_counter();
        satp = prc.get_table_address() >> 12;
        pid = prc.get_pid() as usize;
      },
      ProcessState::Sleeping => {

      },
      _ => {},
      }
    }
    println!("Scheduling {}", pid);
    PROCESS_LIST.replace(pl);
    if frame_addr != 0 {
      // MODE 8 是 39 位虚拟地址 MMU
      // 我使用 PID 作为地址空间标识符，
      // 希望在我们切换进程时帮助（不？）刷新 TLB。
      if satp != 0 {
      return (frame_addr, mepc, (8 << 60) | (pid << 44) | satp
  );
      }
      else {
      return (frame_addr, mepc, 0);
      }
    }
    }
  }
  (0, 0, 0)
```

```
}
```

这不是一个好的调度程序，但它可以满足我们的需要。在这种情况下，调度程序返回的只是运行进程所需的信息，每当我们执行上下文切换时，我们都会询问调度程序并获得一个新进程，有可能得到完全相同的过程。

您会注意到，如果我们没有找到进程，我们会返回 (0, 0, 0)，这实际上是此操作系统的错误状态，我们将需要至少一个进程（init）。在这里，我们将 yield，但现在，它只是循环通过系统调用将消息打印到屏幕上。

```
// 我们最终会将这个函数移出这里，
// 但它的工作只是在进程列表中占据一个位置。
fn init_process() {
  // 在我们有系统调用之前，我们不能在这里做很多事情，
  // 因为我们是在用户空间中运行的。
  let mut i: usize = 0;
  loop {
    i += 1;
    if i > 70_000_000 {
      unsafe {
        make_syscall(1);
      }
      i = 0;
    }
  }
}
```

A.4.5 切换到用户

```
.global switch_to_user
switch_to_user:
  # a0 - 帧地址
  # a1 - 程序计数器
  # a2 - SATP 寄存器
  csrw    mscratch, a0

  # 1 << 7 is MPIE
  # 由于用户模式为 00，
  # 我们不需要在 MPP 中设置任何内容（bits 12:11）
  li    t0, 1 << 7 | 1 << 5
  csrw  mstatus, t0
  csrw  mepc, a1
  csrw  satp, a2
  li    t1, 0xaaa
  csrw  mie, t1
```

```
  la     t2, m_trap_vector
  csrw   mtvec, t2
  # 这个 fence 强制 MMU 刷新 TLB。
  # 然而，由于我们使用 PID 作为地址空间标识符，
  # 我们可能只在创建进程时才需要它。
  # 现在，这确保了正确性，但它不是最有效的。
  sfence.vma
  # A0 是上下文帧，所以我们需要重新加载它
  # 并 mret 以便我们可以开始运行程序。
  mv  t6, a0
  .set  i, 1
  .rept 31
    load_gp %i, t6
    .set  i, i+1
  .endr

  mret
```

当我们调用这个函数时，我们不能期望重新获得控制权，那是因为我们加载了我们想要运行的下一个进程（通过它的陷入帧上下文），然后我们在执行 mret 指令时通过 mepc 跳转到该代码。

A.4.6 整合起来

那么，这如何结合在一起呢？好吧，我们将来某个时候会发出一个上下文切换计时器。当我们遇到这个陷入时，我们调用调度程序来获取一个新进程，然后我们切换到该进程，从而重新启动 CPU 并退出陷入。

```
7 => unsafe {
  // 这是上下文切换计时器。
  // 我们通常会在这里调用调度程序来选择另一个进程来运行。
  // 机器定时器 Machine timer
  // println!("CTX");
  let (frame, mepc, satp) = schedule();
  let mtimecmp = 0x0200_4000 as *mut u64;
  let mtime = 0x0200_bff8 as *const u64;
  // QEMU 给出的频率是 10_000_000 Hz，
  // 因此这会将下一个中断设置为从现在开始一秒后触发。
  // 这对于正常操作来说太慢了，但它能让我们看到屏幕后发生的事情。
  mtimecmp.write_volatile(mtime.read_volatile() + 10_000_000);
  unsafe {
    switch_to_user(frame, mepc, satp);
  }
},
```

我们再次缩短了 m_trap 函数，但是请看一下陷入处理程序，我们每次都重置

内核栈，这对于单个 hart 系统来说很好，但是当我们进行多处理时我们必须更新它。

A.4.7 结论

启动一个进程并不是什么大不了的事，但是它要求我们暂时放下我们曾经想的编程方式。我们正在调用一个函数（switch_to_user），这将使 Rust 不再起作用，但它仍然可以工作?!为什么，好吧，我们正在使用 CPU 来改变我们想要去的地方，Rust 是不明智的。

现在，我们的操作系统处理中断和调度进程，当我们运行时，我们应该看到以下内容 A.3！

```
Set stack from 0x0000000100000000 -> 0x000000008025c000
Set stack from 0x0000000100001000 -> 0x000000008025d000
Init's frame is at 0x8025b000
Init process created at address 0x80024da0
UART interrupts have been enabled and are awaiting your command.
Getting ready for first process.
Issuing the first context-switch timer.
Scheduling 1
Scheduling 1
Test syscall
Scheduling 1
Test syscall
Scheduling 1
QEMU: Terminated
```

图 A.3 中断和调度进程运行图

每当我们执行上下文切换计时器时，我们都会看到“调度 1”，现在是每秒 1 个，这对于普通的操作系统来说太慢了，但它给了我们足够的时间来看看发生了什么。然后，进程本身 init_process 在 70,000,000 次迭代后进行系统调用，然后将“Test syscall”打印到屏幕上。

我们知道我们的进程调度程序正在运行，并且我们知道我们的进程本身正在 CPU 上执行。因此，我们成功了！

综合论文训练记录表

学生姓名		学号		班级	
论文题目					
主要内容以及进度安排	指导教师签字： 考核组组长签字： 年　　月　　日				
中期考核意见	考核组组长签字： 年　　月　　日				

指导教师评语	指导教师签字：_____ 年　　月　　日
评阅教师评语	评阅教师签字：_____ 年　　月　　日
答辩小组评语	答辩小组组长签字：_____ 年　　月　　日

总成绩：_____

教学负责人签字：_____

年　　月　　日