**目录**

[摘要 2](#_Toc1877764862)

[第1章 绪论 3](#_Toc445112230)

[1.1研究背景 3](#_Toc83662982)

[1.2研究意义 4](#_Toc86680837)

[1.3研究内容 5](#_Toc1876647301)

[第2章 RISC-V 指令集详述 5](#_Toc1756987572)

[2.1 RISC-V概述 6](#_Toc1263065115)

[2.2 RISC-V 设计规范 7](#_Toc1349127135)

[2.2.1指令子集 7](#_Toc1452205124)

[2.2.2 用户级指令 8](#_Toc1856468976)

[2.2.3 特权级 9](#_Toc976342328)

[2.2.4 状态和控制寄存器 10](#_Toc1868491943)

[第3章 基于RISC-V的ucore实现方式 11](#_Toc66313481)

[3.1中断、异常和系统调用部分移植 12](#_Toc229440476)

[3.1.1 x86架构下中断、异常和系统调用的处理过程 12](#_Toc1287215933)

[4.1.2 RISC-V架构下中断、异常和系统调用的处理过程 13](#_Toc356296989)

[3.2 内存管理部分移植 15](#_Toc766772926)

[3.2.1 RISC-V内存管理机制 15](#_Toc1825241104)

[3.2.2 物理内存管理 18](#_Toc1518949926)

[3.2.3虚拟内存管理 22](#_Toc1665829385)

[3.3进程/线程管理部分移植 23](#_Toc1035279493)

[3.3.1 线程控制块 23](#_Toc860011191)

[3.3.2 线程的创建 24](#_Toc1521514308)

[3.3.3 进程退出和等待进程 25](#_Toc1911404725)

[3.3.4 进程的切换 25](#_Toc1593990207)

[第4章 实验结果和测试 26](#_Toc1225618333)

[4.1实验结果测试 26](#_Toc147405045)

[4.2 实验结果分析 26](#_Toc1888571949)

[参考文献 27](#_Toc658496643)

[致谢 27](#_Toc32532279)

[附录 27](#_Toc1051444637)

**摘要**

**第1章 绪论**

## 1.1研究背景

指令集是软件和硬件之间沟通的桥梁，不同的厂家可能会用不同的方法来实现相同的指令集，从而让遵循相同指令集的软件可以无需修改即可运行。比如我们熟知的Intel和AMD都实现了x86指令集，这基本上统治了PC的市场。而出现较晚的ARM指令集则基本占领了移动端处理器的大部分市场

早期的微处理器大部分为复杂指令集（CISC）架构，即设计尽可能复杂的指令来完成任务，原因之一是当时的编译技术并不发达。而随着编译器技术以及其他技术的发展，研究人员渐渐发现多数的复杂指令只在很少的时候被用到，而且复杂的指令限制了处理器速度的进一步提高。精简指令集（RISC）技术应运而生。IBM 801可能是第一个用精简指令集的理念来设计的系统，并发展成为了今天的Power架构。1980年左右Berkeley的Dave Patterson主导了Berkeley RISC项目并设计了其第一代的处理器RISC I，这就是RISC这个名称的由来。Sun Microsystem（现在的Oracle）的SPARC处理器架构也有很多Berkeley RISC影子在里面。差不多同时，斯坦福的教授John Hennessyq EER和他的团队也从课程设计出发设计了MIPS处理器的早期雏形，我们能在很多家用路由器里见到他的影子。而ARM，迄今为止发展的最好的RISC指令集，已经无处不在。

但是现有的各种指令集依旧不能满足所有人的要求，Intel的X86是CISC指令集，过于复杂和庞大，而且存在专利问题。而ARM除了专利问题外，若想自行设计基于ARM指令集的处理器，需要非常昂贵的License。OpenRISC作为一个开源的指令集架构，其许可证为GPL，这意味着所有的指令集改动都必须开源。而且，OpenRISC发展缓慢，设计过于老旧，64位架构也不成熟。

在这样的背景下，一个名为RISC-V的开源指令集应运而生，它是一个完全开放的、标准的、能够支持各种应用的新指令集，并且它基于一个非常宽松的名为BSD License的许可证发布：任何公司都可以在自己的产品中免费使用，而修改也无需再开源，只要最终致谢作者即可。

RISC-V的优越特性使其获得了众多高校和大公司的关注，并将其应用的手机芯片、物联网、大数据等各个方向，拥有良好的发展前景。

## 1.2研究意义

开放源代码是不可逆转的趋势，RISC-V指令集是一个现代的、没有专利问题和没有历史包袱的全新指令集，并且以BSD许可证发布。和成就了ARM的授权模式相比，RISC-V不仅能让公司收益，也能够让大学和研究机构更好地研究新的处理器技术和架构。

与成熟的ARM指令集相比，RISC-V不紧不需要高昂的授权费，还具有安全性高的优点:RISC-V源代码的完全开放可以确保最高的信任级别，更加符合关心产品安全的用户的需求，特别是希望基于RISC-V进行一些开发并用于商业用途的用户。这样一个简单、低功耗的CPU设计对学术界和工业界都有很重要的意义。

最初只是由加州大学伯克利分校开发用于计算机教学的RISC-V处理器架构渐渐得到了产业界的支持。三星、AMD、Intel、NVida、Marvell、Synopsys、高通、IBM等公司都积极参与RISC-V的相关工作中。

现在已经有非常多有意义的基于RISC-V的项目出现，Berkeley自己开发了多款开源的处理器，可以覆盖从高性能计算到嵌入式等应用领域。RISC-V正在逐渐走向成熟，并且随着很多高校和大公司的加入，我们有理由相信，它将成为互联网世界不可或缺的重要角色。

ucore是一个学生学习操作系统用的比较精简的OS，参考了MIT用于学生教学的操作系统xv6。与xv6一样，ucore也是基于x86指令集的，ucore的构建是一个循序渐进的过程，让学生参与到ucore的构建过程中，尝试着一步一步完成一个从“空空如也”到“五脏俱全”的“麻雀”操作系统，包含虚存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能，在这个过程中加深对操作系统原理和实现细节的理解。Ucore总的内核代码量（C+asm）不会超过5K行。充分体现了“小而全”的指导思想。

但是由于x86指令集是CISC指令集，本身就比较复杂，并且Intel公司在x86体系架构发展的每一个阶段都保持[向后兼容](http://baike.baidu.com/item/%E5%90%91%E5%90%8E%E5%85%BC%E5%AE%B9/94553" \t "http://baike.baidu.com/_blank)的特性，使其实现更加复杂，甚至有些实现逻辑为了向后兼容而变得很奇怪，不易理解。这大大加大了学生学习操作系统的难度，因为学生在学习操作系统相关内容时，不得不花费很多精力关注底层硬件细节。RISC-V指令集的精简性和完全开源的特性非常适用于教学，故将用于教学的ucore操作系统从x86架构移植到RISC-V架构是非常有意义的工作。可以让学习者在充分了解和学习操作系统原理和实现细节的过程中，可以减少对硬件底层的不必要的关注，从而提高学习的效率。

## 1.3研究内容

本文完成的主要工作是将基于X86指令集的操作系统分ucore移植到RISC-V指令集。论文主要包括五部分:

第一章简略介绍了RISC-V指令集和ucore操作系统，阐述了本文选题的背景和意义。该章在论文中起到了引入和铺垫的作用。

第二章详细介绍了RISC-V指令集，包括RISC-V的历史和特点、设计规范等。了解RISC-V是移植工作的前提。

第四章介绍了ucore移植的细节，其中包括中断与系统调用移植、内存管理移植、线程和进程管理移植等部分。是本文的主要工程目标。

第五章是论文的结束语，其内容主要有：对本文相关工作的总结和对系统下一步工作的展望。

1. **RISC-V 指令集详述**

**2.1 RISC-V概述**

RISC-V是基于已建立的RISC原则的开放指令集架构。与大多数ISA相比，RISC-V ISA可以自由并且免费地用于任何目的，任何人都可以设计，制造和销售RISC-V芯片和软件。虽然它不是第一个开放式ISA，但却意义重大，因为他被设计为能够在现代计算机领域使用，如云计算、物联网、手机芯片等。这些使用场景需要设计者考虑性能和功率。并且作者在设计指令集的同时还提供了大量的支持软件和工具链，避免了很多新指令在产生指出不能为用户提供足够支持的问题。

RISC-V指令集的产生基于一定的历史背景。主流芯片架构（英特尔和ARM）都受专利保护，有严格的许可证规则，需要昂贵的授权费。但是即便付了授权费，指令集也十分复杂，需要很大的工作量才能让其适应具体的工作需求。更何况，有一些小型的公司或者研究者想要为自己特定的应用开发自己的芯片，他们没有足够的资金支付昂贵的授权费，有些研究者希望可以公开分享自己的工作，却不得不考虑这样做是否会违反许可条款。所以需要有一个完全不受限制的并且简洁的指令集来解决这些问题。在这种历史背景下，RISC-V应运而生。

RISC-V指令集架构起源于加州大学伯克利分校的，最初是这所大学为了帮助学生学习计算机架构而开发的。随着它的发展，创建者们[希望](http://www.eeworld.com.cn/tags/%E5%B8%8C%E6%9C%9B" \t "http://www.eeworld.com.cn/qrs/2014/0822/_blank)将它推向主流，帮助推动云计算和[物联网](http://www.eeworld.com.cn/IoT/" \t "http://www.eeworld.com.cn/qrs/2014/0822/_blank)等新兴市场。目前加州大学伯克利分校已经基于RISC-V创造了几个芯片，其他的一些机构还有多个项目正在进行之中。

事实上，还有其他的开源指令集，包括OpenRISC和SPARC V8。但是RISC-V的一些特性证明它更适合当前的发展趋势。物联网、云计算和个人移动设备是未来计算机技术中的重要领域，RISC-V的设计与这几个领域的需求非常契合：

1. 基本加扩展的指令集体系结构[8]：由于不同领域具有不同的特性，所以对指令集也有不同的需求，基本加拓展的指令集体系结构可以为不同领域定制一套自己专有的指令集以提高效率和降低成本。为了配合这种需要，RISC-V指令集体系结构包括一个足以支持编译器和操作系统的小的核心指令集和标准但可选的扩展指令集用于帮助片上系统实现针对不同应用的定制看，并且为全新指令操作码保留了空间。

2.紧凑的指令集编码：物联网设备对成本敏感，需要采用较小的内存空间和规模较小的代码。

3.在单精度和双精度的基础上，同时支持四倍精度的浮点运算：一些在当今数据仓库规模计算机上运行的应用由于所处理的巨大数据集的要求，已经使用支持四倍精度浮点运算的软件库。

4.在32位和64位寻址的基础上，同时支持128位寻址：物联网设备有限的内存容量意味着32位寻址在未来的几十年内仍然重要，而64位寻址已经成为其他更大设备上的事实标准。虽然今天的数据仓库规模计算机行业还不需要2128字节，但我们有理由相信，不超过十年，数据仓库规模计算机就可能需要比264更多的字节（1600亿亿字节或16艾字节）来在其所有的固态非易失性存储器寻址。指令集体系结构支持的地址位数不够是一种很难纠正的错误[9]，因此最好现在就规划较长的地址。

RISC-V这个新的指令集架构（ISA），最初旨在支持计算机体系结构研究和教育，现在，它将成为RISC-V基金会管理下的逐步应用于工业界的标准开放架构。

**2.2 RISC-V 设计规范**

**2.2.1指令子集**

RISC-V定义了一个基本的整数 ISA，必须在任何实现中存在，另外可以包含基于基本ISA的其他扩展。这个基本核心被小心地限制具有最少的指令，足够支持一个合理的目标机，以便编译器、汇编器、链接器、操作系统（包含额外的管理员级操作）可以在之上运行，这样就可以提供一个方便的ISA和软件工具链“骨架”，围绕它可以构建更为定制化的处理器ISA。

每一个基本整数指令集，被整数寄存器宽度和相应的用户地址空间大小进行分类。有两种主要的基本整数变种， RV32I 和 RV64I。未来计划支持128位地址空间。  
 基本整数 ISA 可被一个或者多个可选指令集扩展进行增强，但是基本整数指令集不能被重新定义。RISC-V指令集扩展分为标准扩展和非标准扩展。标准扩展一般都是有用的，并且与其它的标准扩展并不冲突。非标准扩展是高度特殊化的，并可能与其它的标准扩展或者非标准扩展冲突。指令集扩展根据基本整数指令集宽度不同，可能有轻微的功能差异。

标准扩展提供乘法/除法、原子操作以及单精度、双精度浮点算术。基本整数 ISA 被命名为“I”其中包含了整数计算指令、整数 load、整数 store 和控制流指令，并且在所有RISC-V 实现中，都是必须的。标准整数乘法和除法扩展被命名为“M”，其中增加了对保存在整数寄存器中的值进行乘法和除法的指令。标准原子指令扩展被命名为“A”，其中增加了对存储器进行原子的读、修改和写操作的指令，以支持处理器间的同步。标准单精度浮点扩展，被命名为“F”，增加了浮点寄存器、单精度计算指令、单精度 load 和 store 指令。标准双精度浮点扩展，被命名为“D”，扩展了浮点寄存器，并增加了双精度计算指令、 load和 store 指令。一个基本整数内核加上这四个标准扩展（“IMAFD”），被缩写为“G”，它提供了一个通用的标量指令集。RV32G 和 RV64G 也就是RISC-V的标准拓展。

除了基本整数 ISA 和标准扩展之外，很少有一条新指令对所有应用程序巨大的好处，虽然它可能在某些领域中非常有用。由于能耗效率要求更为特殊化，我们相信对于一个 ISA 规范中的必须部分的简化是很重要的。鉴于其他的体系结构通常将它们的 ISA 作为一个单一的整体，它们会随着时间推移，当加入新指令的时候，就变化到一个新的版本。然而 RISC-V尝试随着时间的推移，保持基本内核和每一个标准扩展不变，相反的，将新指令作为可选的扩展。

**2.2.2 用户级指令**

由上文可知，RISC-V有多种指令子集可以选择，考虑到ucore是32位的操作系统，本小节选择RV32I基本整数指令集为例来解析RISC-V的用户级指令。RV32I核心指令主要包括以下几类：

**整数计算指令**，绝大多数整数计算指令对保存在整数寄存器中的XLEN位值进行操作。整数计算指令要么为寄存器-立即数操作，要么寄存器-寄存器操作。对于寄存器-立即数指令和寄存器-寄存器指令，其目标都是寄存器rd。

**控制转移指令**，RV32I提供了两类控制转移指令：无条件跳转和条件分支。无条件跳转指令包括跳转并连接指令JAL和间接跳转指令JALR，都使用pc相对寻址，以便于支持位置无关代码。条件分支指令被设计为在两个寄存器之间进行算术比较操作，而不是像x86一样使用条件码。因为作者认为一条组合了比较和分支的指令，可以很好地适应常规的流水线，避免了使用额外的条件码状态或者使用临时寄存器，减少了静态代码大小、降低了动态指令取指通信量

**内存访问指令，**RISC-V是一个load-store体系结构，也就是说，只有load和store指令可以访问存储器，而算术指令只在CPU寄存器上进行操作运算。Load和store指令在寄存器和存储器之间传输数值为了获得最高的性能，所有load和store指令的有效地址，应该与该指令对应的数据类型相对齐。基本ISA支持非对齐的访问，但是根据实现的不同，这可能会运行得非常慢。

**控制和状态寄存器指令，**系统指令用于访问那些可能需要特权房改问的系统功能，原子性读-修改-写控制和状态寄存器（CSR）的指令是一种系统指令。主要包括CSRRW（Atomic Read/Write CSR）、CSRRS（Atomic Read and Set Bits in CSR）、CSRRC（Atomic Read and Clear Bits in CSR）等。

**环境调用和断点，**ECALL指令用于向支持的运行环境发出一个请求，这个运行环境通常是一个操作系统。EBREAK指令被调试器所使用，用来将控制权传送回给调试环境

**2.2.3 特权级**

正如x86架构提供了从ring0到ring3四种特权级来对机器进行保护，RISC-V架构也提供了四种特权级，不过RISC-V架构和x86架构的特权级并不是一一对应的，两者的设计思路完全不同。

RISC-V定义的四种特权级如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 级别 | 编码 | 名字 | 缩写 |
| 0 | 00 | User | U |
| 1 | 01 | Supervisor | S |
| 2 | 10 | Hypervisor | H |
| 3 | 11 | Machine | M |

机器级是最高级特权，也是 RISC-V 硬件平台唯一必须的特权级。运行于机器模式（M-mode）下的代码是固有可信的，因为它可以在低层次访问机器的

实现。用户模式（U-mode）和管理员模式（S-mode）被分别用于传统应用程序和操作系统，而Hypervisor模式（H-mode）则是为了支持虚拟机监视器。

所有硬件实现必须提供M-mode，因为这是唯一的模式，可以不受限制地访问整个机器。最简单的RISC-V实现可以仅提供M-mode，虽然这样做不能为防止不正确的、恶意应用代码提供保护。许多RISC-V实现还支持至少一个用户模式（U-mode），以对系统的其他部分进行保护，防止被应用程序代码破坏。管理员模式（S-mode）可被加入，以在管理员级操作系统和管理员执

行环境、硬件抽象层之间提供隔离。

基于x86架构的ucore操作系统用到了ring0和ring3两个特权级，所有的内核代码都在ring0上执行，所有的用户代码都在ring3上执行，从而实现对各类资源的保护。基于RISC-V架构的ucore需要设计M-mode、S-mode和U-mode三个特权级。U-mode用来执行用户级的代码，M-mode和S-mode用来执行内核代码。事实上，内核代码主要放在S-mode，只有非常少的跟底层硬件相关的代码需要陷到M-mode执行，可以说，M-mode与x86架构的部分硬件功能对应,S-mode与x86架构的ring0对应。

这与RISC-V架构设计者的设计意图是一致的：M-mode是用于低层次的访问一个硬件平台，是上电复位后进入的第一个模式。M-mode也被用于实现那些硬件太难直接实现或者耗费太大的特性。而管理员模式被故意地限制了与底层物理硬件的交互，例如物理存储器和设备中断，以支持清晰的虚拟化，便于操作系统的移植。

**2.2.4 状态和控制寄存器**

所有的控制和状态寄存器是与某个特级相关的，但是它们页可以从更高的特权级进行访问。下面将列出一些比较重要的CSR。

|  |  |
| --- | --- |
| 名字 | 描述 |
| matstus | 机器状态寄存器 |
| mtvec | 机器自陷处理函数基地址 |
| mtdeleg | 机器自陷转移寄存器 |
| mie | 机器中断使能寄存器 |
| mtimecmp | 机器墙钟定时器比较值 |
| mtime | 机器墙钟时间寄存器 |
| mtimeh | mtime 的高 32 位，仅 RV32 |
| mscratch | 机器自陷处理函数Scratch寄存器 |
| mepc | 机器异常程序计数器 |
| mcause | 机器自陷原因 |
| mbadaddr | 机器坏地址 |
| mip | 机器挂起的中断 |
| mtohost | 到主机去的输出寄存器 |
| mfromhost | 从主机来的输入寄存器 |

|  |  |
| --- | --- |
| 名字 | 描述 |
| status | 管理员状态寄存器 |
| stvec | 管理员自陷处理函数基地址 |
| sie | 管理员中断使能寄存器 |
| stimecmp | 墙钟（ Wall-clock）定时器比较值 |
| stime | 管理员墙钟时间寄存器 |
| stimeh | stime 的高 32 位，仅 RV32 |
| sscratch | 管理员自陷处理函数 Scratch 寄存器 |
| sepc | 管理员异常程序计数器 |
| scause | 管理员自陷原因 |
| sbadaddr | 管理员坏地址 |
| sip | 管理员挂起的中断 |
| sptbr | 页表基地址寄存器 |

1. **基于RISC-V的ucore实现方式**

ucore是专门为教学用的一个精简的操作系统，故设计了lab1-lab8，每个lab完成操作系统的一部分功能，从零开始，循序渐进一步步构建一个完整的操作系统。

lab1完成操作系统的启动，主要实现了一个可以切换到x86保护模式，能够读取磁盘并加载ELF执行文件格式，并显示字符的bootloader和一个可以处理时钟中断和显示字符的简易操作系统。

Lab2完成了物理内存管理，实现了一个简单的物理内存管理系统。主要包括发现系统中的物理内存，即连续物理内存管理，和通过建立页表来实现虚拟内存到物理内存之间的映射三部分。

Lab3完成了虚拟内存管理，在lab2的基础上，借助于页表机制和中断异常处理机制，完成Page Fault异常处理和FIFO页替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用 。

Lab4完成了内核线程管理，主要实现了内核线程的创建和调度。包括设计管理线程的数据结构即进程控制块、为新创建的内核线程分配资源、进程切换等核心内容。

Lab5完成了用户进程管理，主要实现了进程的创建、执行、切换和退出，并且实现了完整的系统调用。Lab4完成了内核线程，至此，所有的运行都在内核态执行。 实验5将创建用户进程， 让用户进程在用户态执行， 且在需要ucore支持时， 可通过系统调用来让ucore提供服务。

Lab6完成了调度器。实验五完成了用户进程的管理，可在用户态运行多个进程。但采用的调度策略是很简单的FIFO调度策略。Lab6设计了一个系统调度器框架，实现了基于此框架的Round-Robin（RR）调度算法。然后参考RR调度算法的实现，完成Stride Scheduling调度算法

Lab7实现了同步互斥。如果多个进程需要协同操作或访问共享资源，则存在如何同步和有序竞争的问题。Lab7实现了进程同步机制—信号量（semaphore） 机制以及基于信号量的哲学家就餐问题解决方案。并参考信号量机制，实现基于管程的条件变量机制和基于条件变量来解决哲学家就餐问题。

Lab8实现了文件系统。ucore模仿了UNIX的文件系统设计，文件系统架构包括通用文件系统访问接口层、文件系统抽象层、Simple FS文件系统层、外设接口层四个部分。并基于此结构实现了文件系统的相关功能。

本文的目标是将ucore操作系统从x86架构移植到risc-v架构，其中ucore操作系统的启动、物理内存管理、虚拟内存管理、内核线程管理和用户进程管理部分与底层硬件密切相关，不同架构下需要不同的实现方法，是移植时需要重点关注的部分。而调度器、同步互斥和文件系统基本上是平台无关的，在移植中不需要大的改动。所以本文工作主要关注的是lab1-lab5几个部分的移植。本章将从以下几个方面详细分析ucore从x86架构移植到risc-v架构的过程：

1. 中断、异常和系统调用
2. 内存管理，包括物理内存管理和虚拟内存管理
3. 进程/线程管理，包括内核线程和用户线程

## 3.1中断、异常和系统调用部分移植

**3.1.1 x86架构下中断、异常和系统调用的处理过程**

中断是指由cpu外部设备引起的外部事件，如I/O中断、时钟中断；异常是指cpu在执行指令时检测到非法的操作而产生的事件，如除数为零、地址访问越界；系统调用是指用户程序调用相关指令请求系统服务产生的事件。在基于X86架构的ucore中，这三种情况的处理方式是一致的，都通过中断机制来处理。

处理思路是当cpu收到中断或者异常时，就会暂停当前正在执行的程序，跳转到相对应的处理例程中，在完成处理后再返回到刚才被打断的程序继续执行。也就是说操作系统如果要正确处理各种中断事件，就需要安排应该由那个中断服务例程负责处理特定的中断事件。系统将所有的值嗯段时间统一进行了编号，这个编号成为中断向量。x86设计了一个中断描述符表（IDT），中断描述符表把中断或者异常编号和指向中断服务例程的描述符建立一一对应关系。这是x86特有的逻辑，RISC-V并没有这种设计。

下面详细解析基于x86的ucore的中断处理过程。

由中断向量找到中断服务例程：cpu从总线上读取到了中断请求对应的中断向量，并以此为索引，到中断描述符表中找到对应的中断描述符，中断描述符中保存着中断服务历程的段选择子。根据IDT中查找到的段选择子可以从GDT中找到对应的段描述符，由此就可得到中断服务例程的起始地址。

CPU需要确认是否发生了特权级的转换：如果发生了特权级的转换，需要进行栈的切换工作。CPU会从当前程序的TSS段里取得该程序的内核栈地址，即包括内核态的ss和esp的值， 并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈 紧接着就将当前程序使用的用户态的ss和esp压到新的内核栈中保存起来。

保存当前被打断的程序现场：保存现场也就是保存寄存器的值，以便于将来返回被打断的程序时可以继续执行。相关的现场信息保存在内核栈中。

CPU将中断服务例程的第一条指令的地址加载到cs和eip寄存器中，开始执行中断服务例程。

在所有中断处理工作完成之后，要通过iret指令恢复被打断的程序的执行。iret指令会自动完成很多工作，它首先从内核栈中取出被打断的程序的现场信息，恢复寄存器；如果存在特权级的转换，还需要从内核栈中取出用户态的ss和esp，把栈切换会原来使用的用户态的栈。

通过以上五个步骤，x86架构下的中断处理基本完成了。

**4.1.2 RISC-V架构下中断、异常和系统调用的处理过程**

在解析RISC-V架构下中断、异常和系统调用的处理过程之前，我们需要先了解一下RISC-V相关的设计和寄存器。自陷被认为是在一个RISC-V线程中出现了一个异常的情况，导致将控制同步传输到自陷处理函数。自陷处理函数通常是在一个更高特权环境中执行的。中断认为是在当前RISC-V线程外异步出现了一个事件。基本上，中断和异常都会转换为自陷来进行处理，如果出现了一个必须处理的中断，将会选择某条指令来接收中断异常，然后顺序地产生一个自陷，异常是否和如何转变为自陷的，依赖于执行环境，虽然预期是绝大多数环境在一个异常被触发时，采取一个精确的自陷。

在x86的逻辑中，中断、异常和系统调用对应不同的情况，但处理方式是一致的，RISC-V的逻辑类似，中断、自陷和异常对应于不同的情况，但处理方式是一致的，最终处理都归于自陷处理。所以RISC-V的自陷可以分成两类，提升特权基级别的自陷称为垂直自陷，而保持在同样特权级别的自陷称为水平自陷。RISC-V特权体系结构提供了将自陷灵活地路由到不同的特权层的机制。

在x86中，有中断向量和中断描述符表IDT的概念，无论是中断异常还是系统调用，都是由中断描述符表路由。RISC-V中没有中断描述符表这个概念，所以需要依赖一些专有的寄存器编写代码来实现路由。

RISC-V与自陷相关的指令和寄存器：

● mtvex寄存器：机器自陷向量基址寄存器，保存自陷处理程序的入口地址

● mscratch寄存器：被用来保存一个指向机器模式硬件线程本地的上下文空间的指针。

● mepc寄存器：机器异常程序计数器，当处理一个自陷时，mepc被写入碰到异常的那条指令的虚地址。中断和其他自陷区分，可以通过一个在mcause 寄存器值符号位上的分支指令来完成。Exception Code可扩展为一个自陷向量表的索引。

● mcause寄存器：机器原因寄存器，最高位为Interrupt位，表示是否为中断引起的异常，如果是的话该位为1，最低四位为Exception Code位，保存了最近一次异常的标识代码。

● mbadaddr 寄存器：机器坏地址寄存器，当出现一个取指地址未对齐异常、取指访问异常、load/store地址未对齐异常、load/store访问异常时， mbadaddr寄存器被写入导致失效的地址。

● mtime、mtimecmp寄存器：机器定时寄存器，通过这两个寄存器可以产生时钟中断。mtime是实时计数器，当mtime寄存器的低32位与mtimecmp寄存器的低32位相同时，将阐释一个定时器中断，这个中断将持续存在，直到通过写mtimecmp寄存器才被清除。

● ECALL指令：ECALL 指令被用于向更高特权级发起请求，执行一条ECALL 指令将导致一个环境调用异常。

● ERET指令：当处理完一个自陷后，ERET指令被用于返回到自陷产生的特权级。

● MRTS指令：MRTS指令将特权模式改变为S，并且将 pc设置为管理员的自陷处理函数，（这个自陷处理函数入口）被保存在stvec寄存器中。另外，mepc、mcause、mbadaddr 寄存器中的值被分别复制到 sepc、scause、sbadaddr寄存器中。与之类似的还有MRTH，机器重定向自陷到 H-mode。

RISC-V中，在缺省情况下，任何特权级的自陷，都是在机器模式下被处理的。处理程序的入口地址保存在RISC-V专有的寄存器机器自陷向量基址寄存器mtvex中。但是，根据上文对RISC-V四个特权级的分析可知，把所有工作放在M-mode来做是不合理的，所以RISC-V提供了自陷重定向指令MRTS将自陷处理从M-mode转移到S-mode。所以一个简单的实现是把所有自陷定向到一个M-mode的处理函数，如果目标是更低的特权级就进行重定向。在现阶段的ucore中就采用了这种实现方式，把自陷处理函数放在M-mode，定义了一个函数trap\_entry，所有自陷发生后都跳到trap\_entry执行。

trap\_entry部分主要完成以下工作：根据mcaue寄存器判定是异常还是中断，并进行相应的处理，在RISC-V中，系统调用又叫环境调用异常，是异常的一种。

如果是中断，则跳到中断处理函数，判断是软中断还是硬中断。然后进行相应的处理，中和x86架构一样，整个过程也是保存上下文（寄存器）、对中断进行处理、恢复上下文。

如果不是中断，就是异常，我们以系统调用这个环境调用异常为例来分析。系统调用通过ECALL指令来实现，在执行ECALL指令之前，会根据RISC-V架构的调用约定，将系统调用相关的参数（如系统调用号等）保存至相应的寄存器，执行ECALL指令后，代码会跳转到统一的自陷处理函数trap\_entry中执行。在trap\_entry部分，首先要将32个通用寄存器的值压栈，然后将mcause、mepc和sp这三个寄存器的值存入通用寄存器a0、a1、a2中，再跳转到系统调用处理函数中。a0、a1、a2这三个寄存器的值是系统调用处理函数的参数，系统调用处理函数根据已经保存入栈的系统调用的参数和mcause、mepc、pc的值针对不同的系统调用做相应的处理。

在上面关于RISC-V架构下的自陷处理进行解析时，我们忽略了可能产生的用户态和内核态切换的问题，在后续出现用户态线程之后，在用户态产生的系统调用都会导致用户态和内核态之间的切换，在此单独阐述一下。在x86架构中，用户态和内核态的切换是硬件自动实现的。首先，在切换到某个用户进程的时候，操作系统会将tss的esp0设置为该进程的内核栈地址。在该用户进程利用INT进行系统调用的时候，会自动地将栈地址切换为tss的esp0的地址并将原来用户栈的地址放在内核栈中，这样便切换到了内核栈。而当需要从系统调用返回的时候，ucore会调用iret指令，iret指令非常智能，当它发现权限（保存于cs，ds段寄存器中）发生变化的时候，会自动地从栈中读取esp的值赋值给esp寄存器，这样，就从内核态又切换回了用户态。在RISC-V中硬件并不能自动完成相关的工作，需要编写操作系统的人自己完成。所以在RISC-V架构中，ucore需要先判断系统调用是来自用户态还是内核态，如果来自用户态，则需要执行一段额外的栈的切换代码，主要是把用户栈的栈顶指针保存到内核栈中，然后把栈顶指针指向内核栈栈顶，内核栈栈顶指针一直被保存在中断帧中。从内核态回到用户态需要取出用户栈栈顶指针，并让当前栈指针指向它。

**3.2 内存管理部分移植**

**3.2.1 RISC-V内存管理机制**

在进行内存管理移植之前，需要先了解RISC-V内存管理机制。

RISC-V为设计者提供了多种可选的虚拟化管理方案，由matstus寄存器中的虚拟化管理字段VM指定，每个VM字段的设置定义了在所有被支持的特权级下面的行为，某些VM设置的行为可能依据硬件所支持的特权级而有所不同.表x.x给出了当前定义好的虚拟化方案。

|  |  |  |  |
| --- | --- | --- | --- |
| 值 | 缩写 | 所需要的模式 | 描述 |
| 0  1  2 | Mbare Mbb Mbbib | M  M，U M，U | 没有翻译或者保护 单个基址和边界 分离的指令和数据基址和边界 |
| 3-7 | 保留 |  |  |
| 8  9 10 11 12 | Sv32 Sv39 Sv48 Sv57 Sv64 | M，S，U M，S，U M，S，U M，S，U M，S，U | 基于页面的32位虚拟寻址 基于页面的39位虚拟寻址 基于页面的48位虚拟寻址 保留给基于页面的57位虚拟寻址 保留给基于页面的64位虚拟寻址 |
| 13-31 | 保留 |  |  |

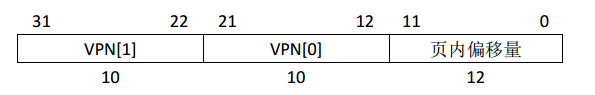
Mbare 对应于没有存储器管理或翻译，因此所有的有效地址，无论其特权模式，都被认为是机器物理地址。Mbare模式是复位时进入的模式。对于一个 RISC-V硬件实现，只有Mbare模式是强制要求的。Mbb是一种“基址和边界”体系结构，它需要系统至少有两个特权级（U和M）。Mbb适用于那些在用户模式代码需要低开销翻译和保护的系统，其不需要复杂的基于页面的虚拟存储管理。Sv32是一个针对RV32系统的基于页面的虚拟存储器体系结构，提供了一个 32 位虚拟地址空间，是RISC-V中的32位页式寻址。Sv39和Sv48是针对RV64系统的基于页面的虚拟存储器体系结构，提供了一个39位或者48位的虚拟地址空间。

ucore选择了Mbare和Sv32两种虚拟化管理方式，当运行在Macheine态时，使用Mbare虚拟化方案，逻辑地址直接对应物理地址。当运行在S态及以上时，选择Sv32虚拟化方案。因为基于x86架构的ucore是一个32位的操作系统，采用32位页式虚拟内存管理，与Sv32对应。

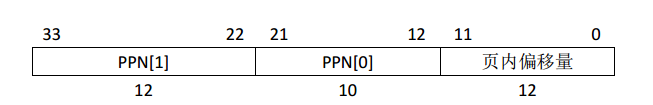
**Sv32详解**

Sv32实现支持一个32位虚拟地址空间，虚拟地址通过一个两级页表  
被翻译为物理地址，页大小为4KB。一个 Sv32 虚拟地址被切分为一个虚拟页编号（virtual page number ，VPN）和页内偏移量（offset），如图 **x。x** 所示。20位的VPN被翻译为一个22位的物理页编号（physical page number，PPN），而12位的页内偏移量不被翻译。

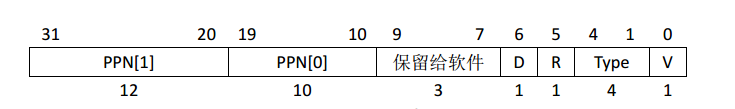
一个Sv32虚拟地址的结构如下：



一个Sv32物理地址的结构如下：



一个Sv32页表项的结构如下：



Sv32 页表包含个页表项（PTE），每个页表项为 4 个字节。一个页表刚好是一个页的大小，并且永远对齐到页的边界。根页表的物理地址被保存在 sptbr 寄存器中。  
 V位表明PTE是否是有效的。Type字段表明了PTE是一个指向下级页表的指针，还是一个叶子 PTE如果是后者，Type 字段同时还编码了访问权限。R位是被引用位，当一个虚拟页被读、写或者取（fetched from）时，实现将设置对应PTE的R位为1。D位是脏位，当一个虚拟页被写时，实现将额外地设置对应PTE的D位为1。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | 含义 | Global | R | W | X | R | W | X |
| 0  1 | 指向下一级页表的指针 指向下一级页表的指针——全局映射 | √ | -- | | | | | |
| 2  3  4  5  6  7 | 管理员只读，用户读-执行页面 管理员读写，用户读-写-执行页面 管理员和用户只读页面 管理员和用户读-写页面 管理员和用户读-执行页面 管理员和用户读-写-执行页面 |  | √  √  √  √  √  √ | √  √  √ | √  √ | √  √  √  √  √  √ | √  √  √ | √  √  √√ |
| 8  9  10  11 | 管理员只读页面 管理员读-写页面 管理员读-执行页面 管理员读-写-执行页面 |  | √  √  √  √ | √  √ | √  √ |  |  |  |
| 12  13  14  15 | 管理员只读页面——全局映射 管理员读-写页面——全局映射 管理员读-执行页面——全局映射 管理员读-写-执行页面——全局映射 | √  √  √  √ | √  √  √  √ | √  √ | √√ |  |  |  |

表 4.2： PTE Type 字段编码

**Sv32虚拟地址翻译过程**

一个虚拟地址va被翻译成一个物理地址pa，如下所示：  
 1.设sptbr的值为a，设i＝LEVELS-1。（对于Sv32，LEVELS等于2）  
 2.设地址a＋va.vpn[i]×PTESIZE处PTE的值为pte。（对于Sv32，PTESIZE等于4）  
 3.如果pte.v＝0，停止并报告一个地址错误。  
 4.否则，pte.v＝1。如果pte.type≥2，继续到步骤5。否则，这个PTE是一个指向下级页表的指针。令i＝i－1。如果i＜0，停止并报告一个地址错误。否则，令a＝pte.ppn×PAGESIZE，转到步骤2。（对于Sv32，PAGESIZE等于212）  
 5.找到一个叶子PTE。通过pte.type判断所请求的存储器访问是否允许。如果不允许，停止并报告一个地址错误。否则，翻译成功。设pte.r等于1，并且如果存储器访问是一个store，设pte.d等于1。翻译后的物理地址如下：pa.pgoff＝va.pgoff。如果i＞0，则这是一个超页翻译并且pa.ppn[i－1:0]＝va.vpn[i－1:0]。pa.ppn[LEVELS－1:i]＝pte.ppn[LEVELS－1:i]

**3.2.2 物理内存管理**

基于x86架构的ucore实现中，物理内存管理部分主要包括物理内存探测、以页为单位管理物理内存和实现分页机制。

**物理内存探测**

在操作系统被启动之后，它需要了解整个计算机系统中的物理内存是如何分布的，哪些可用，哪些不可用。在基于x86架构的ucore中，通过BIOS中断来获取内存布局，具体是利用参数为e820h的INT 15h的中断来实现。BIOS通过系统内存映射地址描述符（Address Range Descriptor） 格式来表示系统物理内存布局， 其具体表示如下：

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 8字节 | base address #系统内存块基地址 |
| 08h | 8字节 | base address #系统内存块基地址 |
| 10h | 4字节 | type of address range #内存类型 |

ucore通过调用INT 15h BIOS中断，让BIOS帮我们查找出一个一个的内存布局，并放入到一个保存地址范围描述符结构的缓冲区中，供后续的ucore进一步进行物理内存管理。缓冲区结构定义如下

struct e820map {  
 int nr\_map;  
 struct {  
 long long addr;  
 long long size;  
 long type;  
 } map[E820MAX];  
};

通过BIOS中断来获取内存信息是x86特有的逻辑，RISC-V中并没有相关实现，但是获取内存布局是必要的，是后续内存管理工作的基础。为了不必大幅度修改ucore代码，我们手动构造了一个e820map，并为其赋值。x86架构为了向后兼容和历史原因，内存布局非常的复杂，RISC-V并不像x86一样，它的内存布局可以非常的简单规整。我们认为RISC-V架构中，内存可以分为两部分，第一部分是已经被内核占用的内存，另一部分是空闲的连续可用的空间。这样我们就可以根据编译链接的结果得知系统的物理内存分布。此外，RISC-V还提供了用于探测物理内存的ABI sbi\_query\_memory。

**以页为单位管理物理内存**

在获得可用物理内存范围后，系统需要建立相应的数据结构来管理物理页。ucore定义了一个page数据结构来表示一个物理页。Page数据结构定义如下，它包含了映射此物理页的虚拟页个数ref，描述物理页属性的flags和双向链接各个Page结构的page\_link双向链表 。

struct Page {

int ref; // page frame's reference counter

uint32\_t flags; // array of flags that describe the status of the page frame

unsigned int property;// the num of free block, used in first fit pm manager

list\_entry\_t page\_link;// free list link

};

在初始情况下，也许这个物理内存的空闲物理页都是连续的，这样就形成了一个大的连续内存空闲块。 但随着物理页的分配与释放， 这个大的连续内存空闲块会分裂为一系列地址不连续的多个小连续内存空闲块， 且每个连续内存空闲块内部的物理页是连续的。 那么为了有效地管理这些小连续内存空闲块。所有的连续内存空闲块可用一个双向链表管理起来，便于分配和释放，为此定义了一个free\_area\_t数据结构，包含了一个list\_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr\_free。 其中的链表指针指向了空闲的物理页。free\_area\_t数据结构定义如下

typedef struct {

list\_entry\_t free\_list; // the list header

unsigned int nr\_free; // # of free pages in this free list

} free\_area\_t;  
 有了这两个数据结构， ucore就可以管理起来整个以页为单位的物理内存空间。根据e820map探测到的物理内存分布，我们可以知道一共需要多少个页来管理物理内存空间，需要为这些页对应的page结构分配出内存空间。由于bootloader加载ucore的结束地址（用全局指针变量end记录）以上的空间没有被使用，所以我们可以把end按页大小为边界去整后， 作为管理页级物理内存空间所需的Page结构的内存空间。至此，物理内存被分成两部分，一部分存放ucore内核和page结构，应该设定为已占用的物理内存空间，另一部分为剩余的内存空间，设定为可用空闲物理内存。对这两部物理内存空间对应的的page页做相应的标记即可完成以页为单位的物理内存管理。

至此，我们已经将所有的物理内存用设定了对应的page结构来管理，相应的，我们也必须实现物理内存分配和回收算法，ucore实现了fistfit内存分配算法，为此设计了一个用来维护一个查找有序空闲块的数据结构free\_area\_t，包括成员结构

list\_entry\_t free\_list; // the list header 空闲块双向链表的头

unsigned int nr\_free; // # of free pages in this free list 空闲块的总数（以页为单位）

可以说这个数据结构的核心依旧是一个双向链表结构，通过对双向链表的插入、删除等操作即可实现物理内存的分配和回收。具体实现细节不在赘述。

无论实在x86架构上还是在RISC-V架构上，以页为单位管理物理内存的实现思路是一致的，只需要在代码细节处进行小的修改。

**实现分页机制**

完成了以页为单位的物理内存管理之后，需要实现分页机制。实现分页机制主要包括两部分，一部分是建立好虚拟地址与物理地址之间的映射关系，即正确建立二级页表，另一部分是使能页机制。

由于ucore OS是基于80386 CPU实现的， 所以CPU在进入保护模式后， 就直接使能了段机制， 并使得ucore OS需要在段机制的基础上建立页机制。这使得页机制的实现非常复杂，基于x86的ucore从计算机加电，启动段式管理机制， 启动段页式管理机制，在段页式管理机制下运行这整个过程中，虚地址到物理地址的映射产生了多次变化，可分四个阶段，每个阶段的映射关系都不一样。RISC-V没有段机制，所以不需要先建立段机制再建立页机制，所以它的映射关系一直不变，非常简单，这是它的一个优点。

**建立二级页表**

无论x86架构还是RISC-V架构，建立页表的流程都是一致的：

1. 先获取一个空闲物理页，用于页目录表
2. 调用相关函数在虚拟地址和物理地址之间建立一一映射关系，具体的处理过程以页为单位进行设置，根据虚拟地址去寻找物理地址，如果页目录项中的存在位为0，表示缺少对应的页表空间，则先获取一个空闲物理页给页表，然后填写页表项的内容。

由于页表项的不同，在x86架构中填写的页目录项内容为：

*页目录项内容 = (页表起始物理地址 &0x0FFF) | PTE\_U | PTE\_W | PTE\_P*

在x86架构对于页表中以线性地址la的中10位为索引值对应页表项的内容为：

*页表项内容 = (pa & ~0x0FFF) | PTE\_P | PTE\_W*

*在RISC-V架构中填写的页目录项内容为：*

页目录项内容 = (页表起始物理地址 &0x0FFF) | PTE\_TYPE\_TABLE | PTE\_V | PTE\_R

在x86架构对于页表中以线性地址la的中10位为索引值对应页表项的内容为：

页表项内容 = (pa & ~0x0FFF)| PTE\_TYPE\_URWX\_SRWX | PTE\_R | PTE\_V

ucore 的内存管理经常需要查找页表：给定一个虚拟地址，找出这个虚拟地址在二级页表中对应的项。通过更改此项的值可以方便地将虚拟地址映射到另外的页上。相关函数在x86和RISC-V中的实现是基本一致的。

**使能页机制**

在页表建立好之后，就可以使能页机制了。在x86架构中，使能页机制主要包括两部分：

1.把页目录表的起始地址存入CR3寄存器中；

2.把cr0中的CR0\_PG标志位设置上。

完成这两部工作之后，计算机就进入了分页模式，但是正如上文所述，x86是在段机制的基础上实现的页机制，映射关系比较复杂，所以在使能页机制之后，还需要更新GDT等操作，由于相关内容太过复杂且并非本文重点，不再详述。

在RISC-V架构中，使能页机制也包括两部分;

1. 把页目录标的起始地址存入sptbr寄存器中。
2. 将操作系统转到s模式，因为页式内存管理只在S特权有效。

完成这两部分工作之后，计算机就进入了分页模式。

**3.2.3虚拟内存管理**

虚拟内存管理是一个操作系统中必要的部分，通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为按需分页；把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出。ucore虚拟内存管理分别实现了页访问异常处理、FIFO页替换算法和页换入换出来完成以上三点需求。

在物理内存空间的管理中有关内存的数据结构和相关操作都是直接针对实际存在的资源，没有从一般应用程序对内存的“需求”考虑，即需要有相关的数据结构和操作来体现一般应用程序对虚拟内存的“需求”。程序运行中访问内存产生页访问异常时， 需要判定这个引起异常的虚拟地址内存访问是越界、写只读页的“非法地址”访问还是由于数据被临时换出到磁盘上或还没有分配内存的“合法地址”访问。因此需要一定的数据结构来描述不在物理内存中的“合法”虚拟页。为此，ucore建立了mm\_struct和vma\_struct数据结构。vma\_struct描述了应用程序对虚拟内存“需求”，是一个合法地址的范围。mm\_struct是一个比vma\_struct更高的抽象层次的数据结构，这个数据结构表示了包含所有虚拟内存空间的共同属性。

**页访问异常处理**

实现虚存管理的一个关键是页访问异常处理，比如，在程序的执行过程中由于某种原因（页框不存在/写只读页等）而使CP 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射时，CPU会产生一次页访问异常， 从而需要进行相应的页访问异常的中断服务例程。 这个页访问异常处理的时机被操作系统充分利用来完成虚存管理，即实现“按需调页”/“页换入换出”处理的执行时机。当相关处理完成后，页访问异常服例程会返回到产生异常的指令处重新执行， 使得应用软件可以继续正常运行下去。产生页访问异常的原因主要有：

1.目标页帧不存在（页表项全为0，即该线性地址与物理地址尚未建立映射或者已经撤销)；

2.相应的物理页帧不在内存中（页表项非空， 但Present标志位=0， 比如在swap分区或磁盘文件上)

3.不满足访问权限(此时页表项P标志=1， 但低权限的程序试图访问高权限的地址空间，或者有程序试图写只读页面).

页访问异常是异常的一种，所以它的处理方式和上文关于异常的解析一致。我们需要关注的是页访问异常处理的主要函数，它根据页访问异常的物理地址以及异常的类型来查找此地址是否在某个VMA的地址范围内以及是否满足正确的读写权限，如果在此范围内并且权限也正确， 这认为这是一次合法访问，但没有建立虚实对应关系。所以需要分配一个空闲的内存页，并修改页表完成虚地址到物理地址的映射，刷新TLB，然后返回到产生页访问异常的指令处重新执行此指令。如果该虚地址不在某VMA范围内，则认为是一次非法访问。

X86与RISC-V关于页访问异常的处理在思路上是一致的，不同之处在于两种架构下异常处理不同，以及由于页表结构不同，导致在权限处页表修改等地方细微的差别。

FIFO页替换算法和页换入换出在本次移植中没有实现，因为使用的工具链暂时不支持硬盘管理，故相关内容也不再详述。

**3.3进程/线程管理部分移植**

线程可以分为内核线程和用户线程，它们的区别有两个：内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的ucore内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的用户内存空间。内核线程和用户线程管理有很多相同的地方，故把他们放在一起分析。

因为涉及的都是单线程，故在本文中不区分线程和进程。

**3.3.1 线程控制块**

线程控制块是用来管理线程的数据结构。如果要让线程运行，首先要创建内核线程对应的进程控制块，还需把这些进程控制块通过链表连在一起，便于随时进行插入，删除和查找操作等进程管理事务。线程控制块结构如下：

struct proc\_struct {

enum proc\_state state; // Process state

int pid; // Process ID

int runs; // the running times of Proces

uintptr\_t kstack; // Process kernel stack

volatile bool need\_resched; // need to be rescheduled to release CPU?

struct proc\_struct \*parent; // the parent process

struct mm\_struct \*mm; // Process's memory management field

struct context context; // Switch here to run process

struct trapframe \*tf; // Trap frame for current interrupt

uintptr\_t cr3; // the base addr of Page Directroy Table(PDT)

uint32\_t flags; // Process flag

char name[PROC\_NAME\_LEN + 1]; // Process name

list\_entry\_t list\_link; // Process link list

list\_entry\_t hash\_link; // Process hash list

};

其中有几个比较重要的成员变量：

● mm：内存管理信息，包括内存映射列表、页表指针等。对内核线程来说，mm没用，设置为0.

● state：进程所处的状态

● parent：用户进程的父进程

● context：进程的上下文，用于进程切换

● tf：中断帧的指针

● kstack：线程的内核栈

● static list\_entry\_t hash\_list[HASH\_LIST\_SIZE]：所有进程控制块的哈希表

● list\_entry\_t proc\_list：所有进程控制块的双向线性列表

**3.3.2 线程的创建**

无论是RISC-V架构还是x86架构，内核进程的创建流程是一致的：

1.分配并初始化进程控制块

2.分配并初始化内核栈

3.根据clone\_flag标志赋值或者共享进程内存管理结构

4.设置进程运行和调度锁虚的中断帧和上下文。

5.把设置好的进程控制块放入hash\_list和proc\_list两个全局进程链表中进行管理

虽然内核创建的流程是一致的，但中断帧和上下文主要是各种寄存器，是平台相关的，所以不同架构下，中断帧的上下文的结构是不同的。比如在x86中，上下文结构中用eip保存切换后要执行的下一条指令，在RISC-V中则用ra来报存。中断帧需要保存中断返回后要执行的指令、栈指针、使能中断相关的寄存器等，这些在不同的架构下对应不同的寄存器。

在x86架构和RISC-V架构上，用户线程的创建流程也是一致的，都是采用内核线程创建用户线程的方式。内核线程首先用上述创建内核线程的方法创建一个用户线程，再通过把应用程序执行码覆盖到该用户线程的用户虚拟内存空间来完成用户线程的创建。这样实现的基础是在编译时，先把应用程序独立编译成一个elf格式的执行码，并在链接时把这个执行码放在编译好的kernel后面，被bootloader一起加载进来。

把应用程序执行码覆盖到该用户线程的用户虚拟内存空间的过程是通过系统调用实现的。具体的工作包括清空用户内存空间mm，解析elf格式的应用程序执行码，建立对应的vma结构并插入到mm架构中，根据应用程序段的大小分配物理空间，并建立对应的页表。此外还要设置用户栈，为用户栈建立vma、申请物理地址，建立页表。建立好mm结构之后，需要修改页表基地址寄存器，更新用户进程的虚拟内存空间，此时用户进程创建基本完成。最后还需要修改中断帧，使系统调用返回后能跳到用户线程入口。

**3.3.3进程退出和等待进程**

进程在执行完自己的工作之后，需要执行退出操作，释放占用的资源。Ucore分了两步来完成这个工作，进程本身完成大部分资源的占用内存回收工作， 然后由此进程的父进程完成剩余资源占用内存的回收工作。

进程退出是通过一个系统调用完成的，具体流程是进程本身释放mm和页表所占用的资源，然后将页表切换为内核态的页表。剩下的资源需要父进程回收，如果当前 进程的父进程处于等待子进程状态，则唤醒父进程，让父进程帮助自己完成最后的资源回收。至此，这个进程完全退出并释放了所有占用的空间，此时，应该调用调度函数，选择新的进程执行。

**3.3.4 进程的切换**

除了上面进程完全退出，还可能出现进程的切换。在x86架构和RISC-V架构中，进程切换的流程是一致的：先找到一个待切换的进程，切换页表，然后进行上下文的切换。上下文切换就是保存旧进程的上下文，恢复新进程的上下文，上下文中保存着切换后要执行的下一条指令。在ucore中，进行上下文切换是会把context中保存的下一个进程要执行的指令地址context.eip放到了堆栈顶，这样在执行上下文切换函数中的最后一条指令“ret”时，会把栈顶的内容赋值给EIP寄存器，这样就切换到下一个进程执行了。

context.eip指向了一个名为forkret的函数，在x86中，forkret函数的作用是找到当前进程的中断帧，并跳转到中断帧中eip保存的指令地址去继续执行中断前的工作。在RISC-V中，forkret函数需要区分是内核线程的切换还是用户线程的切换，如果是内核线程，就执行ret指令，跳转到中断帧的tf->eip执行。如果是用户线程，就涉及到一个特权级的切换问题，因为上下文切换的完整过程是用户态自陷到内核态，在内核态完成上下文切换，再返回用户态，所以此时需要借助回到自陷产生的特权级的指令ERET跳转到用mepc寄存器保存的返回用户态后应该继续执行的指令的地址。ERET指令被设计为在返回特权级的同时，将pc设置为Xepc寄存器保存的值，此处X是ERET指令被执行时的特权模式。

为什么x86架构下foekret函数不需要区分是用户态进程的切换还是

内核态进程的切换呢。因为x86的中断帧中保存了CS、DS段寄存器，能够区分是用户段还是内核段。

1. **实验结果和测试**

本项目最终的成果是将基于x86架构上的ucore移植到了RISC-V架构上，使其能实现中断、系统调用、内存管理、进程和线程管理等操作系统的基本功能。

**4.1实验结果测试**

如果想要测试试验结果，需要先安装测试环境。RISC-V的作者在创造RISC-V指令集的同时，也提供了相应的仿真器和工具链。RISC-V工具链riscv-tools包括了编译器、模拟器、测试用的宏等，主要组成部分如下：

● riscv-gnu-toolchain ，一个RISC-V交叉编译器

● riscv-fesvr ，用于实现在上位机和CPU之间通信机制的库

● riscv-isa-sim ， 指令集模拟器

● riscv-opcodes ，RISC-V操作码信息和转换脚本

● riscv-pk - 提供一个运行RISC-V可执行文件运行的最简的程序运行环境，同时提供一个最简单的bootloader

● riscv-tests - RISC-V指令集测试用例

仿真器可以选择spike或者qemu，本文选择的是spike1.7版本。

RISC-V工具链的安装可以参考RISC-V官网给出的指导，配置好环境后即可编译测试移植好的基于RISC-V架构的ucore。

**4.2 实验结果分析**

本项目基本实现了将基于x86架构的ucore移植到RISC-V架构上的目标，最终完成了一个可以在RISC-V架构上运行的简易的操作系统。这个操作系统具有中断处理、内存管理、线程管理等操作系统的基本功能，能够运行一些简单的用户程序。但是也有一些问题，一个问题是操作系统的功能不够完善，不能对线程进行良好的调度，不能处理同步互斥问题，没有文件系统，因为这些内容基本上是平台无关的内容，所以在本次移植中没有考虑，但是这是一个完整的操作系统应该具有的功能。另一个问题是代码不够精炼，因为是一个移植的过程，所以很多代码是从x86逻辑强行转换过来的，有很多不必要的代码或者能够用更精炼的、符合RISC-V逻辑的实现。这些问题是后续应该改进的部分。

参考文献

致谢

附录