摘要

UCORE 是清华大学操作系统课采用的教学操作系统，其设计参考了 MIT 的 XV6、Harvard 的 OS161 和 Linux。UCORE 受 Linux 影响最深，可以认为是 Linux 的极简版。为了教学的需要，UCORE 被拆分成了八个实验，它们分别是中断管理子系统，物理内存子系统，虚拟内存子系统，内核线程子系统，用户进程子系统，进程调度子系统，进程间通信子系统和文件系统。由于历史原因，UCORE 是基于 x86 平台的。这使得学生不得不花费较大的精力，学习与操作系统无关的关于过时硬件的知识，比如 8086 模式，A20 地址总线，实模式等。

为了抹平学生在学习道路上这些不必要的障碍，可以将 Ucore 运行在一个指令集精简的、没有历史包袱的现代化的指令集上。本文选择的指令集是 RISC-V 64 位指令集目前已经完成了 Ucore 到 RISC-V 32 位指令集的移植，本文将基于以上工作实现到 RISC-V 64 的移植。由于 Ucore 天然地分成了多个子系统，所以本文也将从各个子系统入手，依次地实现对 Ucore 的支持。其中内存管理子系统和进程管理子系统的移植工作量较大，文件系统的移植难度较高。中断管理子系统，调度子系统，进程间通信子系统移植较为容易。另外，Ucore 目前只支持单核处理器，本文也将尝试设计 Ucore 对多处核理器的支持。

目前已经完成了 Ucore 各个子系统的移植工作，Ucore 可以在 QEMU 虚拟机上成功运行了。另外，本文还分析了 XV6 和 Linux 对多核的支持，对于在 Ucore 上实现多核进行了一些探索。

关键词：Ucore，RISC-V，多核

Porting of Ucore to RISC-V 64bits multi-core processor

SHI Zhenxing (Software Engineering)
Directed by XIA Min

ABSTRACT

The UCORE operating system is the teaching operating system of Tsinghua university. Its design is based on MIT's XV6 , Harvard's OS161 and Linux. In order to meet the demands of teaching , UCORE is subdivided into eight experiments. For historical reasons , UCORE is based on x86 platforms. This makes students have to spend a lot of energy on learning about outdated hardware , such as the 8086 mode , the A20 address bus , the real mode , and so on , regardless of the operating system.

In order to smooth out these unnecessary obstacles on the way of learning, Ucore can be run on a modern instruction set that is reduced in instruction set and without historical baggage. The instruction set selected in this paper is the risc-v 64-bit instruction set, and the transplantation of Ucore to risc-v 32-bit instruction set had been completed yet. This paper will implement the migration of risc-v 64 based on the above work. Since Ucore is naturally divided into multiple subsystems, this paper will also start from each subsystem, and then implement the support for Ucore in turn. The memory management subsystem and the process management subsystem have a large migration workload, and the file system is more difficult to transplant. Interrupt management subsystem, scheduling subsystem, interprocess communication subsystem migration is relatively easy.

Now that the migration of the Ucore subsystems has been completed, Ucore can run successfully on the QEMU virtual machine. In addition, this paper also analyzed the support of XV6 and Linux on multi-core, and made some explorations on realizing multicore on Ucore.

KEY WORDS: Ucore, RISC-V, Multi-core

目录

第一章 引言

本章简要介绍了 Ucore 和 RISC-V，并说明了全文的组织结构。

1.1 研究背景

Ucore 是清华大学操作系统课的教学操作系统，任何人都可以通过学堂在线学习这门课程。Ucore 有两个版本，一个是基本的版本，只保留了操作系统最核心的功能，并且为方便教学被划分成了八个部分，方便学生循序渐进地学习。这八个部分是中断管理子系统，物理内存子系统，虚拟内存子系统，内核线程子系统，用户进程子系统，进程调度子系统，进程间通信子系统和文件系统[1]。另一个版本是 Ucore+，偏重于研究。Ucore+的设计目标是：1，划分一个硬件抽象层以方便对各个平台的支持。2，在各个子系统上都建立了框架，以方便实现不同版本的子系统。3，支持内核动态加载。

RISC-V 是加州大学伯克利分校开发的指令集，由于其开源的特性以及先进的设计而受到了广泛的支持。RISC-V 基金会的成员包括一系列重量级的互联网公司、半导体厂商以及研究机构。可以说，RISC-V 代表了未来硬件设计的发展趋势。RISC-V 共包含有 3 种类型的指令:基本指令，标准扩展指令和特殊扩展指令。基本指令是不可更改，必须被包含的指令，它们都是整数指令。基本指令只有 47 条，它被设计为足以支持现代的操作系统，非常适于教学的需求。标准扩展指令包括乘除指令，原子指令，单精度浮点指令和双精度浮点指令。基本指令和标准扩展指令加起来共五种指令，它们提供了完整的通用标量指令集，是工具链默认支持的指令集。特殊扩展指令只用于特定领域，不适用于全部通用领域[2]。

1.2 研究现状

由于 RISC-V 的以上优点，使其特别适合教学研究。例如，可以在组成原理课上实现一个支持基本指令集的 RISC-V CPU，可以在操作系统课上实现一个支持 RISC-V 的操作系统，可以在编译原理课上实现一个支持 RISC-V 指令集的编译器。而确实也是已经有学生将 Ucore 移植到了 RISC-V 32 位平台上。

1.3 本论文的工作和组织

本实验将在已经移植到 RISC-V 32 平台的 Ucore 基础上，实现到 RISC-V 64 平台的移植，并设计对多处理器的支持。其中对多处理器的支持是难点。如何让多个核即并行又协同，就像武侠小说里的左右互搏之术一样，即充满挑战也威力巨大。本文的组

织结构如下:

第一章为引言，讲述了从 Ucore 移植到多核 RISC-V 的研究背景，以及 Ucore 的移植现状和 RISC-V 提供的工具链支持。

第二章为 Ucore 原理的分析，从编译过程、运行过程和逻辑结构三个方面分析了 Ucore 的内在结构。

第三章是对 RISC-V 指令集和 BBL 的分析。由于操作系统与底层硬件关系密切，所以对底层所能提供的接口和功能的理解是非常重要的。

第四章详述了从 Ucore 到 RISC-V 64 移植的具体过程，从 Ucore 的各个子系统的角度分别描述了移植的过程。

第五章介绍了在 Ucore 上支持多处理器的设计，讲述了对称多处理和不对称多处理两种设计方案。

第六章是对实验结果的验证，以及对移植完毕后系统运行现象的描述。

第七章为总结与展望，讲述了本实验接下来的几种可能的发展方向。

第二章 Ucore 原理分析

Ucore 虽然是经过精简的、用于教学的操作系统，但是其本身的结构依然是比较复杂的。本章尝试从编译过程，运行过程和逻辑结构这三个角度来分析 Ucore，期望能从多个侧面，由浅入深地解释清楚 Ucore 的全貌。

2.1 从编译过程观察 Ucore 的组织结构

使用命令 `make -n > make.txt` 可以得到具体的编译过程，从而可以看到 Ucore 的各个部分是怎么一步步地组合成一个完整的镜像的。

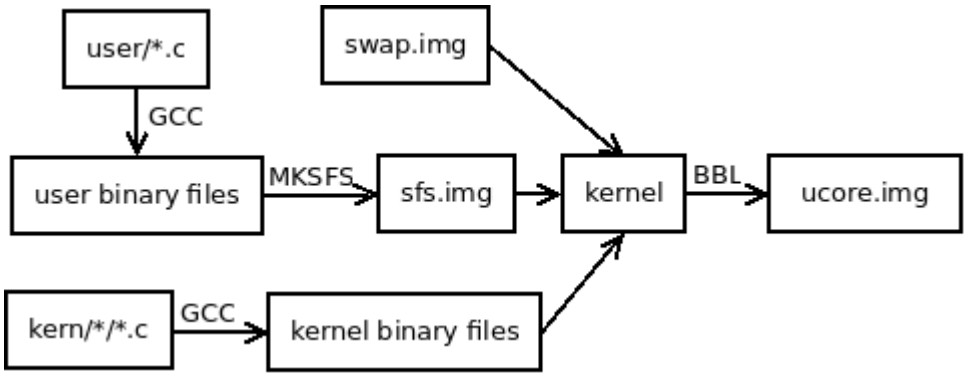


图 2-1 Ucore 编译过程

swap.img 的生成：用 `dd` 命令生成大小为 32k 的 swap.img 文件，里面填充的全为 0。

sfs.img 的编译过程：首先将 `user/libs/*.c` 和 `libs/*.c` 编译为对应的 obj 文件，它们是作为 `user` 目录下其它 c 文件的库而出现的。然后，使用前面编译出的库文件，将 `user/*.c` 编译为对应的可执行文件。最后，使用程序 `mksfs` 将前面的可执行文件依照指定文件系统的格式装载到文件 `sfs.img` 中。

内核二进制文件的编译过程：首先将 `kern/libs` 编译为对应的 obj 文件，它们是作为 `kern` 目录下其它 c 文件的库而出现的。然后，使用 `libs` 目录和 `kern/libs` 目录下的库，将 `kern/*/*.c` 编译为对应的 obj 文件。

kernel 的生成：使用 `ld` 命令将内核二进制文件，swap.img 和 sfs.img 文件链接成为 kernel。并通过 `objdump` 命令生成 **kernel.asm** 和 **kernel.sym**。

ucore.img 的生成：最后一行在 `riscv-pk/build` 目录以 **kernel** 作为 payload 编译生成了 **bbl**，**bbl** 就是最终的 **ucore.img**。

2.2 从运行过程观察 Ucore 的组织结构

tools/kernel.ld 文件指示了 Ucore 的链接规则，从中可以看出 kernel 的入口在符号 kern_entry 处，第一条指令就是从那里开始的。

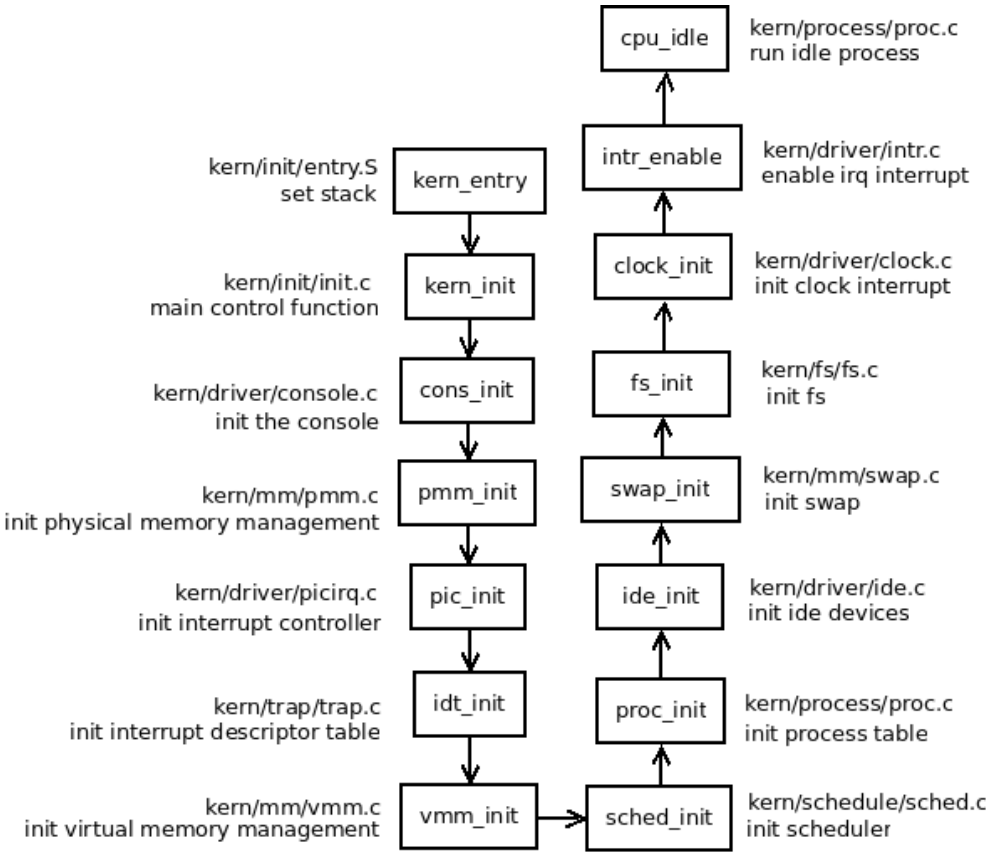


图 2-2 Ucore 运行过程

kern_entry 为系统设置了一个栈，其作用就是为 C 代码的运行准备环境。接下来就是跳到了 C 代码中，内核的总控函数 kern_init。kern_init 主要是完成了两个功能：一是初始化硬件以与底层硬件交互，二是初始化内核的各个子系统以通过系统调用的方式与上层交互。kern_init 完成后内核就进入了调度的状态，等待用户程序的执行。

2.3 Ucore 的逻辑结构

Ucore 中广泛使用了链表来组织内部资源，而且所有的链表都使用了相同的结构。为了使同一种链表结构能访问所有的资源，Ucore 通过计算内存偏移的方法来访问不同的资源。

2.3.1 中断管理子系统

中断管理是操作系统的基础功能，Ucore 操作系统的其它子系统大多依赖中断的功能才得以运行。

trap 函数调用 trap_dispatch 函数，trap_dispatch 函数依据 tf->cause 来判断中断的来源从而确定具体的处理函数。由于 scause 寄存器的最高位为 1 表示中断，为 0 表示异常，所以 tf->cause 小于 0 时调用中断处理程序 interrupt_handler，tf->cause 不小于 0 时调用异常处理程序 exception_handler。系统调用 syscall 发生在异常处理程序中。当中断来源是 CAUSE_USER_ECALL 或 CAUSE_SUPERVISOR_ECALL 时，首先会执行 tf->epc += 4，然后再调用 syscall。tf->epc += 4 使系统调用返回后得以继续执行下一条指令。

syscall 首先从中断帧中读出保存在寄存器 a0 中的系统调用号。然后以系统调用号为索引从系统调用表 syscalls[]中找到对应的系统调用并执行之。最后，当系统调用返回时，把返回值再保存在中断栈中 a0 寄存器对应的位置上。

2.3.2 物理内存管理子系统

物理内存管理的建立是由 pmm_init 函数实现的。

default_pmm.c 里是 Ucore 的物理内存管理算法的实现代码，Ucore 采用了 first fit 算法。free_list 是存放空闲内存块的链表，nr_free 记录了空闲内存页的数量。当有一个内存请求发过来时，会依次遍历 free_list 里的每一项，直到找到一个足够大的内存块，并把它分配出去。如果找到的这个内存块远大于要请求的大小，则会将此内存块拆分，然后把剩下的部分再放入 free_list 里。

2.3.3 虚拟内存管理子系统

虚拟内存管理的本质是让两种访问速度不同的存储介质协同作用，这样的需求是广泛存在的，所以虚拟内存技术有着重要的作用。Ucore 也支持虚拟内存管理。

虚拟内存管理是由 vmm_init， ide_init， swap_init 三个函数实现的。

虚拟内存的机制的实现靠的是缺页中断。当访问的数据不在物理内存中的时候，会触发缺页中断，缺页处理程序将硬盘中的数据调入物理内存，从而实现扩展物理内存的作用。详见 vmm.c 。调用关系：__alltraps->trap->exception_handler->pgfault_handler->do_pgfault

2.3.4 进程管理子系统

进程控制块是进程管理的核心数据结构，其核心的成员变量有：进程状态，父进程，进程上下文的指针，中断帧的指针，页目录的地址，内核栈的指针等。

枚举类型 proc_state 定义了进程的状态，进程的转换过程如图 2-4 所示。

内核创建的第一个线程是 idle，它的任务就是在处理器空闲的时候死等其它进程的创建和执行。第二个内核线程是 init_main，它的作用是创建其它进程。第三个内核

线程是 user_main，它的任务是创建第一个用户进程，第一个用户进程也就是其它用户进程的共同祖先。

其它的用户进程复制自第一个用户进程，而第一个用户进程却不能从内核线程复制，而是需要手工构造出来。第一个用户进程的创建过程：首先创建一个子进程，然后将一个具体的程序放到内存中，接下来通过一个异常调用进入用户态，最后开始执行指定程序的代码。

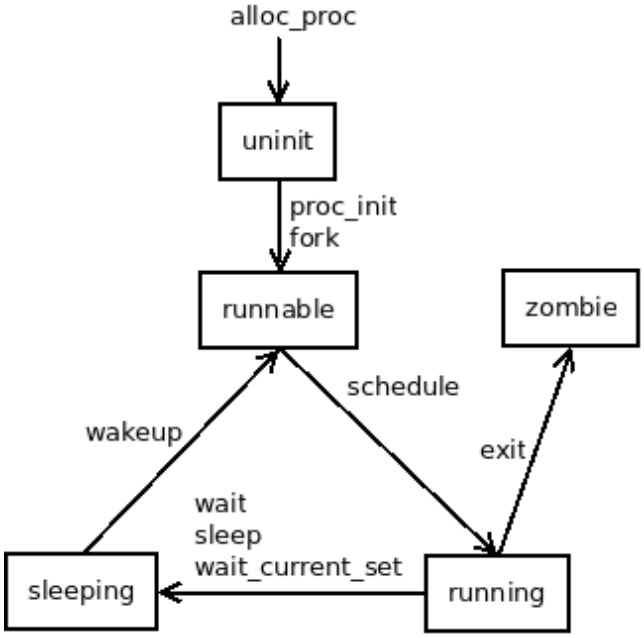


图 2-3 进程状态转换图

2.3.5 调度管理子系统

sched_class 结构体是重要的数据结构。以它为框架可以方便的实现各种处理器调度算法。

shed_init 是对 sched_class->init 的封装。

schedule 函数是 sched_class 调度功能的封装。

switch_to 是上下文切换的代码，在 kern/process/switch.S 文件里。context 结构体决定了上下文切换的次序。switch_to 仅由 proc_run 调用， proc_run 仅由 schedule 调用。

2.3.6 进程间通信子系统

进程间通信是通过同步和互斥来实现的。Ucore 上设计了多种手段来实现同步互斥包括中断，等待队列，信号量和管程。

开关中断来实现同步互斥是最简单的办法。但它有一些无法克服的弊端，一是不适用于多处理器，二是整个系统被挂起的风险很大，三是效率低下。

当进程无法获取临界区的资源时，就会进入 sleeping 状态，接下来 Ucore 就会把进程放入等待队列中。这使得其它处于就绪队列的进程可以占用 CPU，从而提高了系

统的执行效率。

信号量机制则是维护了一个等待队列和一个变量。变量记录了允许交互的进程数量，当进程不满足信号量的条件时就会进入等待队列直到变量条件允许。

2.3.7 文件系统

Ucore 的文件系统划分了四层，它们分别是接口层、抽象层、文件系统层和外 设接口层。如表 2-1 所示：

表 2-1 文件系统分层

名称	表现形式	实现方式
接口层	系统调用	将管道，目录，文件等都抽象为文件描述符
抽象层	抽象函数	屏蔽具体的文件系统实现，为内核其它部分提供统一接口
具体实现层	操作函数	超级块，根目录结点，freemap 区，inode 区
外设接口层	设备抽象	将对设备的操作抽象为对块的操作

第三章 RISC-V 指令集与 BBL 分析

3.1 RISC-V 简介

RISC-V 设计思想是非常先进的。其它处理器的 ISA 是划代的，不同的代实际上就是不同的 ISA 了，每代都有兼容前代的压力，往往会出现一些奇怪的设计。学习者不得不学习那些过时的知识，而设计者也不得不支持那些过时的设计。RISC-V ISA 的设计则是分模块的，基本模块是必须被包含的集合，其被设计为足以支持现代的操作系统，基本扩展模块是大部分场景均会用到的模块，可以使处理器获得更高的效率。而扩展模块则仅适用于特定场景，用以进行特定场景的运算。学习者可以从基本模块开始学习，循序渐进还不必担心版本迭代的问题。设计者只须关注微架构的设计即可，而不必考虑令人生厌的兼容问题[3]。

RISC-V 共有 3 种类型的指令:基本指令，标准扩展指令和特殊扩展指令。基本指令被命名为 I，是不可更改，必须被包含的指令，它们都是整数指令。标准扩展指令包括 M(扩展乘除指令)，A(扩展原子指令)，F(单精度浮点指令扩展)，D(双精度浮点指令扩展)，G(IMAFD 五种指令的总和，提供了完整的通用标量指令集，是默认工具链支持的指令集)。特殊扩展指令只用于特定领域，不适用于全部通用领域。基本指令固定是 32 位的，然而扩展指令可以是变长的，但必须是 16 的倍数[4]。

3.2 寄存器

用户可见的整数寄存器共有 32 个。其中 x0 寄存器是由硬件连线为 0 的，在 X86 中并没有这样的寄存器，然而它的存在有着重要的意义。X0 寄存器主要有两个作用，一是整数 0 的提供者，二是接收不需要的数据。由于 x0 寄存器的以上两个重要作用，使得 RISC-V 的指令集得到大大的精简，只要区区的 47 条指令就可以支持现代操作系统了。其它的 31 个寄存器用于保存各种指针和数据，参阅指令集手册不难理解。

Ucore 所在的权限级别为 S 级，还可以看到 3 类控制和状态寄存器：Trap Setup 、Trap Handling 和 Protection & Translation。

Trap Setup: sstatus 寄存器用于管理中断总体，sie 用于管理详细的中断 。sedeleg 和 sideleg 用于给用户级的异常和中断授权使其可以在用户级处理，否则会转到管理员级来处理，由于节省了特权级转换的开销，这样可以提高效率。stvec 用于提供中断管理程序的入口。scounteren 用于管理对硬件性能计数器的的访问。

Trap Handling: sepc 记录了引发中断的地址，这样中断返回后才能正确执行下一条指令。sscratch 记录了上下文切换的指针，这样中断才能正确切换。scause 记录了引起中断的原因，这样中断管理程序才能找到正确的函数处理此中断。stval 记录了引起错误的内存地址，这样为中断管理程序才能处理这样的内存错误。sip 则描述了当前正在处理什么样的中断。

Protection & Translation: satp 记录了一级页表的地址，这样就能进行地址映射了[5]。

3.3 RV32I 指令集

RV32I 指令集是 RISC-V 指令集的核心与基础。

1. 整数计算指令

包括了算术运算，移位运算和逻辑运算。可以发现在整数计算指令里并没有立即数减法指令，这是由于减法和加法互为逆运算，可以通过加法运算来模拟减法运算，从而精简了指令集。这样的巧思有很多，由此可见 RISC-V 的特色之处。

2. 控制转移指令

包括了无条件跳转和有条件分支两种指令。其中无条件跳转包含了读写寄存器的操作，这与 x86 是不太一样的，这样设计是为了方便函数跳转。而有条件跳转只分等于、不等于、小于和大于等于四种选项，这样的设计也是很巧妙的。因为小于可以方便的转换成大于，大于等于可以方便的转换成小于等于，精简了指令集。

3. LOAD 和 STORE 指令

Load 指令用于把内存中的数据加载寄存器中，Store 指令用于把寄存器中的数据保存到内存中。它们的操作对象有 3 种：8 位的 byte 型数据，16 位的 halfword 型数据和 32 位的 word 型数据。这是 RISC 指令集的特色，只允许特定的指令读写内存，帮助程序员培养良好的编程习惯。

4. 内存模型

内存模型指令的作用是使指令的执行序列化。

5. 控制和状态寄存器指令

控制和状态寄存器记录和控制着处理器的状态，没有它们则无法实现现代处理器的特性。对 CSR 寄存器的操作有 3 种：写、置位和清除位。与 x0 寄存器配合使用可实现对 CSR 寄存器的灵活操作。

6. 环境调用和断点

环境调用指令仅仅是产生一个异常，它并不做其它的事情。断点指令则是用于虚拟机中断点的控制。

3.4 RV64I 指令集

RV64I 是 RV32I 的变体，理解 RV64I 需要以 RV32I 为基础，分析与 RV32I 的不

同之处。

寄存器状态：寄存器扩展为 64 位，且支持 64 位用户地址空间。

整数计算指令(增加了 9 条特有指令)：RV64I 的指令长度也是 32 位的，操作的是 64 位数值。但也提供了变种指令来操作 32 位数值，这些指令的操作码后面都增加了后缀“W"，是 RV64I 特有的指令。增加的指令是 ADDIW, SLLIW, SRLIW, SRAIW, ADDW, SLLW/SRLW, SUBW/SRAW。

Load 和 Store 指令(增加了 3 条指令)：增加 LD(64 位)，LWU(32 位无符号)，SD(64 位）三条指令。

系统指令：与 RV32I 完全相同，只是操作的对象换成了 64 位的。需要注意的是，伪指令 rdcycle，rdtime，rdinstret 分别读取的是相应寄存器的全部 64 位(cycle，time，instret 计数器)，所以 rdcycleh，rdtimeh，rdinstreth 在 RV64I 里就不需要了，是非法的。

3.5 原子扩展指令

原子扩展指令实现的是原子操作，它主要用于实现各种锁，这对于进程间的同步互斥和支持多处理器有重要作用。

3.6 特权级

目前(Priv1.10)， RISC-V 最多有 3 个特权级，0 级最低，3 级最高，2 级保留不用。详见表 3-1。

表 3-1 特权级

级别	名字	缩写	特点
0	用户/应用	U	应用程序
1	管理员	S	操作系统
2	Reserved		
3	机器	M	必需的，可信的，最高级

3.7 对 BBL 的分析

3.7.1 概述

bbl: Berkeley Boot Loader，但它没有完成一个 boot loader 的功能，它只是把普通意义上的内核再打包成了一个新内核，主要还是为操作系统提供了一组调用接口(SBI)。源代码地址：https://github.com/riscv/riscv-pk

fdt : Flattened Device Tree(扁平设备树)，是一个描述系统中硬件的数据结构。操作系统使用 fdt 来查找和注册系统中的设备。它可以描述的硬件包括：CPU 的数量与种类，内存的基址与大小，总线与桥，外围设备的连接，中断控制器与 IRQ 线的连接。它是对硬件的一个封装，减少内核对硬件的依赖，降低内核在设计和编译上的要求 。U-Boot 和 Linux 均使用了扁平设备树这样的动态接口。

htif : Host/Target Interface，它使用前端服务来与 target(sodor)通信。HTIF 是 Berkeley 处理器的非标准工具，因此没有相关的文档。随着 RISC-V 平台规范的发布和内核可以自主更新，它很快就会消失。

16550uart：uart 芯片的一种型号，由美国国家半导体公司生产，并产生了许多变体，包括 16C550， 16C650， 16C750， 和 16C850.

ipi : interprocessor interrupts(处理器间中断)。

hls： hart-local storage， at top of stack

plic： Platform Level Interrupt Controller

clint： Core Local Interrupter，提供实时时钟，计时器和处理器间中断。

3.7.2 从编译过程看 BBL 的组织结构

1. 将 bbl 文件夹下的 logo.c， payload.S， raw_logo.S 编译为库 libbbl.a
2. 将 machine 文件夹下的 fdt.c， mtrap.c， minit.c， htif.c， emulation.c， muldiv_emulation.c， fp_emulation.c， fp_ldst.c， uart.c， finisher.c， misaligned_ldst.c， mentry.S， fp_asm.S 编译为库 libmachine.a
3. 将 util 目录下的 snprintf.c， string.c，编译为 libutil.a
4. 使用以上库将 bbl 目录下的 bbl.c 编译为 bbl.o
5. 依据 bbl.lds 将 bbl.o 编译为 bbl，内核也是在此时加到 bbl 里的

3.7.3 从运行过程看 BBL 的组织结构

函数调用关系：reset_vector->do_reset->init_first_hart->boot_loader->boot_other_hart->kernel

init_first_hart 函数执行过程：

1. query_uart：获取 uart 的信息。
2. query_uart16550：获取 uart16550 的信息。
3. query_htif：获取 htif 的信息。
4. hart_init：初始化当前的硬件线程。

5. hls_init：初始化本地 hart 存储的结构体(struct hls_t)。
6. query_finisher：找到电源按钮，这样 poweroff 函数就可以工作了。
7. query_mem：获取内存的信息。
8. query_harts：获取 harts 的信息。
9. query_clint：设置内核本地中断。
10. query_plic：设置中断控制器。
11. wake_harts：唤醒硬件线程
12. plic_init：初始化中断控制器
13. hart_plic_init：初始化硬件线程的中断控制器
14. memory_init：初始化内存大小
15. boot_loader：调用 bbl/bbl.c 里的 boot_loader 函数

3.7.4 接口：SBI 函数

调用路径：mcall_trap<-trap_table<-.Lhandle_trap_in_machine_mode<-trap_vector<-do_reset

表 3-2 SBI 函数

函数名	意义
sbi_console_putchar	在控制台打印单个字符，无返回。
sbi_console_getchar	返回从控制台获取的字符。
sbi_set_timer	输入 64 位无符号整型，设置计时器，无返回。
sbi_send_ipi	IPI 类型为 IPI_SOFT，向所有处理器发送 IPI 中断。
sbi_remote_fence_i	IPI 类型为 IPI_FENCE_I，向所有处理器发送 IPI 中断。
sbi_remote_sfence_vma	IPI 类型为 IPI_SFENCE_VMA，向所有处理器发送 IPI 中断。
sbi_remote_sfence_vma_asid	只比 sbi_remote_sfence_vma 多输入个参数 asid。
sbi_clear_ipi	无输入，无输出。清除处理器间中断。
sbi_shutdown	关机

第四章 Ucore 到 RV64 的移植

本章讲述 Ucore 从 RISCV32 到 RISCV64 的移植过程。

RISC-V 通过分层的方式明确划分了操作系统与其它层的关系，Ucore 使用了如图 4-1 所示的分层结构。

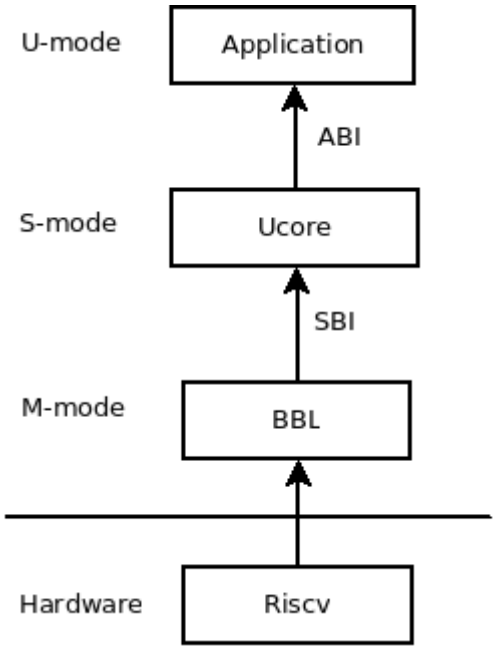


图 4-1 Ucore 与 Riscv 的关系

4.1 Ucore on Riscv32 的重现

首先，需要编译 GCC for Riscv32 的工具链。由于 RISC-V 目前还处于快速的演进期，不同的 spec 版本之间差异巨大。Ucore for Riscv32 是基于最新的指令集版本的 (priv1.10)。在安装工具链的时候要特别注意这一点。

```
git clone https://github.com/riscv/riscv-tools  # 下载工具链的源代码

export RISCV=/path/to/install/riscv/toolchain   # 指定工具链的安装目录
```

```
export PATH=$RISCV/bin:$PATH    # 将工具链加入执行路径
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev
cd riscv-tools
git submodule update --init --recursive     # 递归地更新当前目录下所有子模块
cp build-rv32ima.sh build-rv32g.sh
vim build-rv32g.sh          # 将"rv32ima"改为"rv32g"， 将"ilp32"改为"ilp32d"
chmod +x build-rv32g.sh     # 增加可执行权限
./build-rv32g.sh >& build.log   # 编译工具链，并将编译过程记录在 build.log 文件
中
```

接下来，编译模拟器 qemu for riscv32。可以在上步下载到的工具链源代码里找到 qemu 的源代码。

```
cd riscv-tools/riscv-gnu-toolchain/riscv-qemu
sudo apt install libgtk-3-dev       # 安装支持的库
./configure --target-list=riscv32-softmmu   # 生成 Makefile
make        # 编译
cp riscv32-softmmu/qemu-system-riscv32 $RISCV/bin   # 将模拟器程序放到执
行路径中
```

最后，可以编译并运行 Rcore for Riscv32 了。

```
git clone -b riscv32-priv-1.10 --single-branch
https://github.com/chyyuu/ucore_os_lab.git
cd ucore_os_lab/labcodes_answer
./gccbuildall.sh    # 将各个 lab 下的 ucore 编译成镜像文件
```

```
cd labX          # 进入某个 lab 目录

make qemu        # 在 qemu 上运行
```

至此，Ucore for Riscv32 的工作重现完毕。

4.2 移植环境的搭建

riscv-tools 是支持 RISC-V 的工具链的集合，是基于 RISC-V 开发的必备工具。当下载完毕此工具链后，通过观察 build.sh 脚本可以发现各个子项目的编译都是通过 build_project 命令实现的。可以找到函数 build-project 在 build.common 里的定义，可以发现它主要完成的是 configure && make && make install 这三个命令，且每个项目的目录下均用 build.log 文件记录了编译过程。

可以通过./configure --help 查看 configure 的参数，控制编译的结果。一个好的做法是，源码树、编译树和安装树应在不同的位置。参数里的配置部分(Configuration)主要是控制信息的输出与展示，包括帮助信息。参数里的安装目录(Installation directories)是用来控制安装树的位置，要想达到更精确的控制，可以使用安装目录的微调。参数里的可选功能(Optional Features)主要是控制库的使用与编译。参数里的可选包(Optional Packages)主要是设置基准 riscv ISA 或 ABI。参数里的有影响力的环境变量(Some influential environment variables)主要是指定编译器、编译器选项、连接选项、库和预处理器选项等。

移植环境的搭建过程：

```
# 编译 64 位工具链

export RISCV=/path/to/install/riscv/toolchain
export PATH=$RISCV/bin:$PATH
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev
git clone https://github.com/riscv/riscv-tools.git
cd riscv-tools
git submodule update --init --recursive
./build.sh


# 编译 64 位模拟器

sudo apt install libgtk-3-dev
sudo apt install libsdl2-dev
```

```
cd riscv-tools/riscv-gnu-toolchain/riscv-qemu
./configure --target-list=riscv64-softmmu
make
cp riscv64-softmmu/qemu-system-riscv64 $RISCV/bin
```

4.3 工作量预测

将被移植的 Ucore 不是完全对应着 RISC-V priv1.10，某些指令和寄存器已经过时如 sbadaddr 寄存器已过时，应使用 stval 寄存器。但直接使用 stval 有问题，需要用它的汇编码 0x143 代替。依据指令手册的描述，sfence.vm 指令已被移除了，应使用改进的 sfence.vma 指令。mbadaddr 寄存器已过时，应使用 mtval 寄存器。但直接使用 mtval 不能识别，需要用它的汇编码 0x343 代替。

RV32I 和 RV64I 寄存器的名称和数量均一致，但是长度不同。RV64I 是 64 位的，RV32I 是 32 位的。

RV32I 的指令 RV64I 均具备，且它们的编码是一致的，均为 32 位，不同点是 RV32I 指令操作的数据对象是 32 位的，而 RV64I 指令操作的数据对象是 64 位的 。RV64I 增设的指令均是为了操作 32 位数据而增加的。需要注意的是，伪指令 rdcycle，rdtime，rdinstret 分别读取的是相应寄存器的全部 64 位 (cycle，time，instret 计数器)，所以 rdcycleh， rdtimeh， rdinstreth 在 RV64I 里就不需要了，是非法的。

RV32 的内存管理规约是 Sv32，RV64 的内存管理规约是 Sv39 或 Sv48。它们都是采用多级页表映射的方式实现从虚拟内存地址到物理内存地址的转换的。所不同的是 Sv32 是二级映射，Sv39 是三级映射，Sv48 是四级映射。Sv32 的每个页表项占用 4 个字节，而 Sv39 和 Sv48 的每个页表项占用 8 个字节。Sv32 是 32 位虚拟地址映射到 34 位物理地址，Sv39 是 39 位的虚拟地址映射到 56 位的物理地址，Sv48 是 48 位的虚拟地址映射到 56 位的物理地址。

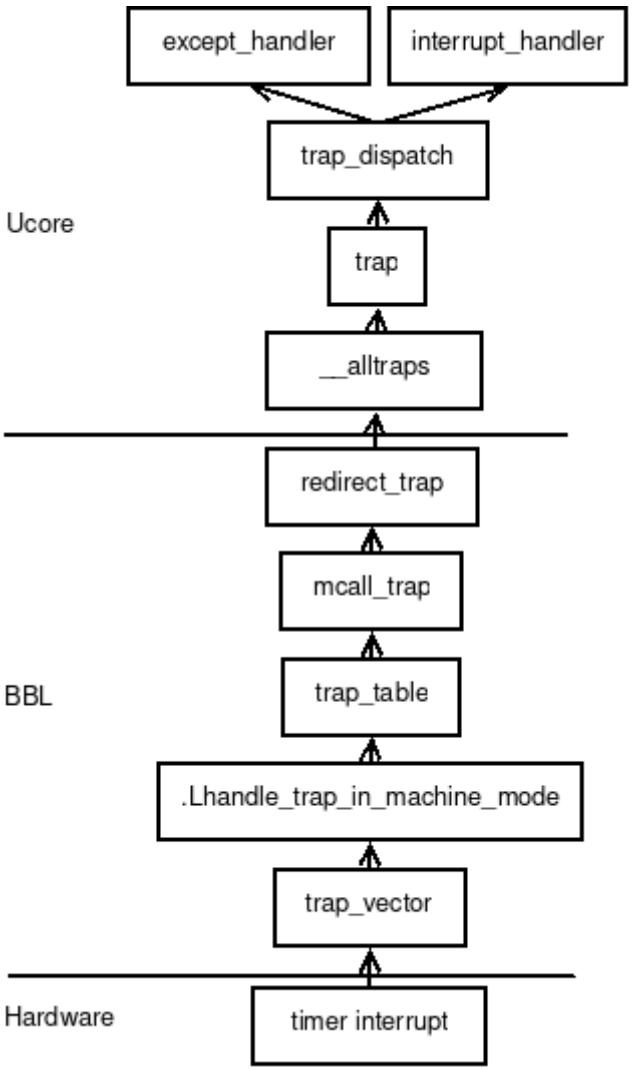
其它的工作量还有地址长度不同，elf 格式不同，BBL 是经过裁减的，RISC-V 的函数参数在寄存器而 X86 在内存，栈里每个单元占的字节数不同。

由于 Ucore 是使用 C 语言开发的，使用的接口也是 SBI，所以移植的时候不用太过关注 64 位和 32 位指令集之间的差异。主要需要考虑的是寄存器里相应位的意义的改变，以及内存管理规约的不同。注：虽然不需要太关注指令集和寄存器的差异，但需要熟悉指令集和寄存器。

4.4 中断子系统的移植

中断子系统是操作系统最基本的功能，其它子系统大多都需要依赖中断子系统才能完成其功能。Ucore lab1 是对中断子系统的实现。

图 4-2 中断子系统调用关系图



1. 修改 Makefile。

变量 GCCPREFIX 的作用是指定所使用的编译器，需要将 riscv32-unknown-elf-修改为 riscv64-unknown-elf-，以指定使用针对 riscv64 的编译器。

变量 QEMU 的使用是指定运行 Ucore 所使用的模拟器，需要将 qemu-system-riscv32 修改为 qemu-system-riscv64，以模拟 64 位的 RISCV 系统。

变量 LDFLAGS 是指定链接器的选项，其中-m 选项的作用是模拟仿真链接器，需要将-m elf32lriscv 修改为-m elf64lriscv。

2. 编译时碰到的第一个问题。

运行 make 命令尝试编译，可以发现在编译到 kern/init/init.c 文件时出现错误，编译过程被强行中止了。其所提示的错误类型为 cast from pointer to integer of different size [-Werror=pointer-to-int-cast]。

为编译过程中碰到问题时有一个明确的解决问题的路线图，有必要对中断子系统使用的底层接口进行一些分析。通过 `grep -r asm` 可以知道在 C 语言里内嵌了哪些汇编指令，通过 `find -name "*.S"` 可以知道哪些文件是用汇编语言写的。由于控制底层接口必然要用到汇编语言，所以只要看系统使用的汇编语言就可以分析出系统使用了哪些底层接口。通过对以上两条命令的结果综合分析，可以看到接口主要由 riscv.h 和 sbi.h 两个文件提供，只有在 clock.c、entry.S 和 trapentry.S 里使用了汇编指令 。clock.c 是在 get_cycles 函数里用了 rdtime 指令来读取 CSR 寄存器 stime 的值 。entry.S 设置了内核栈的地址，为进入 C 程序做准备。trapentry.S 是用 load 和 store 指令来保存和恢复寄存器的值，用于中断的切换。另外，sbi.h 里的函数是通过异常调用的方式由 BBL 里的 mcall_trap 函数实现的。

经过如上分析，再解决编译时问题就有了一个明确的导向。通过参照源文件可以发现，源文件将函数指针和变量指针都指定为 int 型变量，这在 32 位系统下是没有错的，然而在 64 位系统下指针都是 64 位的，再把它指定为 32 位 init 型变量当然会触发变量大小不一致的错误了。更一般地，应该为不同种类的变量指定其特有的变量类型。

在 Ucore 里变量类型定义在 `libs/defs.h` 文件中，通过阅读源文件可以发现应该将指针类型定义为 `intptr_t` 或 `uintptr_t`。但还需要做一点点修改，因为 `defs.h` 文件里直接把指针的类型定义为 32 位的，需要加几条预处理语句先判断一下编译对象的类型，再决定使用 32 位还是 64 位。有一个变量`__riscv_xlen` 指明了当前系统是多少位的，可以用它来判断当前系统的类型。所以，对 defs.h 文件的更改如下：

```
#if __riscv_xlen == 32
typedef int32_t intptr_t;
typedef uint32_t uintptr_t;
#elif __riscv_xlen == 64
typedef int64_t intptr_t;
typedef uint64_t uintptr_t;
#endif
```

在 init.c 文件里，对指针的类型定义都从 `int` 改为 `intptr_t`。此问题得到解决。

3. 编译时碰到的第二个问题。

在编译到 riscv-pk/bbl/logo.c 文件时出现错误，编译过程被强行终止。其提示的错误类型为：**unrecognized argument in option '-mcmodel = medany'**。

然而问题却不在于此，riscv-pk 目录下的 configure 文件，其`--host` 参数的作用是指定交叉编译所在的主机，由于之前传给`--host` 的变量是 `riscv32-unknown-elf`，导

致未能使用正确的编译器，所以才会出现此问题。解决方法就是把 riscv32-unknown-elf 替换为 riscv64-unknown-elf。

4. 运行时碰到的问题。

编译的问题解决后，内核就可以运行起来了。但有时还是会出现问题。其表现为中断不能正常触发。为解决此问题，需要对中断子系统本身的执行流程有一个深入的理解。

kern_init 是 Ucore 内核的总控函数，所以通过对 kern_init 进行分析就可以看出来中断子系统实现了些什么功能。在 kern_int 函数里，cons_init， pmm_init， pic_init 都是空的，也就是 console 、物理内存和中断控制器均未初始化。对 console 的读写是通过 sbi 函数来实现的，而对中断控制器的管理在 BBL 里已经实现了，所以 console 和中断控制器在 Ucore 内核里是不需要初始化的。而物理内存管理是 lab2 的内容，在本阶段是真的没有初始化。cprintf 函数最终是调用 sbi_console_putchar 实现其功能的，其实现的逻辑可以参考 C 语言标准库里的 printf 函数，但有些参数表达的意义略有不同 。idt_init 函数本来是在 X86 处理器下初始化中断向量表的，但在 RISCV 里并没有中断向量表的概念，而是通过一个中断例程来管理所有中断的，所以 idt_init 函数现在的功能就是将中断服务例程(__alltraps)的入口地址保存到 CSR 寄存器 stvec。clock_init 函数控制时钟中断发生的频率并且使能了计时器中断。intr_enable 函数使能了全局中断。综上，中断子系统主要是实现了 cprintf 函数和时钟中断。

中断的过程：

1) __alltraps 在 trapentry.S 里定义，它的地址保存在 stvec 寄存器中，是中断发生时的处理例程。它首先做的是交换 sp 和 sscratch 两个寄存器的值。

2) 如果 sp 的值非 0，则中断来自于用户模式，sp 正指向内核栈，则直接执行第 4 步。如果 sp 的值为 0，则中断来自于内核模式，内核栈的指针保存在 sscratch，需要执行第 3 步。

3) 将 sscratch 寄存器的值保存在 sp(即 x2)寄存器中，使 sp 切换到内核栈。

4) 将除 x2 外的 32 个通过用寄存器(x0~x31)保存到栈中相应的位置，寄存器在栈中的位置由 struct trapframe 定义。

5) 将 sscratch 寄存器置 0，并保存到栈在 x2 寄存器对应的位置上。这使得假如发生中断的嵌套，系统可以正确地返回当前的中断。

6) 将 sstatus， sepc， stval， scause 寄存器保存到栈相应的位置上。

7) 将 sp 寄存器的值保存到 a0 寄存器，作为 trap 函数的参数，然后调用 trap 函数处理中断。trap 返回后，执行 _trapret。

8) 判断 sstatus 寄存器的 SPP 位是否为 0，若为 0 说明来自用户模式，需要执行第 9 步。若非 0 说明不是来自用户模式，直接执行第 10 步。

9) 将之前的内核栈指针保存在 sscratch 寄存器中。

10) 将栈中的内容保存到除 x2 外的 32 个通过用寄存器(x0~x31)。

11) 最后再依据栈中的内容恢复 x2 寄存器的值。

12) 从中断中返回。

经过如上分析，找到了解决此问题的一个方法。就是在调试的时候发现，只要在 trap 函数内放一条 cprintf 语句中断即可正常执行了。这个问题在随后的时间里进行了更细致的分析，并且得到了根本上的解决，请参看本章第 11 节 **Bug** 分析与修复。

得益于 BBL 已经屏蔽了 RV32I 和 RV64I 指令集之间的差异，且对底层硬件进行了封装。所以中断子系统不必考虑指令集的差异和对底层硬件的控制，从而大大简化了移植的过程。对中断子系统的移植集中在对数据类型的修改上。由于 RV64 位下的地址和寄存器是 64 位的，所以它们不是 int 型而是 long 型的变量了。所有使用 int 类型变量来代表内存地址和寄存器的地方均需修改为 long 类型的变量。需要修改的文件有 init.c，sbi.h， printfmt.c， defs.h。

4.5 物理内存管理子系统的移植

内存是 CPU 唯一能访问的大容量存储区域，是所有进程的跑马场。在继中断子系统后，只有做好了内存管理子系统，才能方便地实现其它子系统。

1. 对编译警告的修正。

对物理内存管理子系统的移植需要以中断子系统的修改为基础。运行 make 命令尝试进行编译，编译通过了，但出来两个警告信息。它们提示的都是 **cast from pointer to integer of different size [-Wpointer-to-int-cast]** 。通过观察源代码所在的文件 `kern/mm/pmm.c` 可以发现，这两处都是为指针指定了错误的数据类型。修改方式就是把数据指针指定为 `uintptr_t` 类型。再编译就没有如上警告信息了。

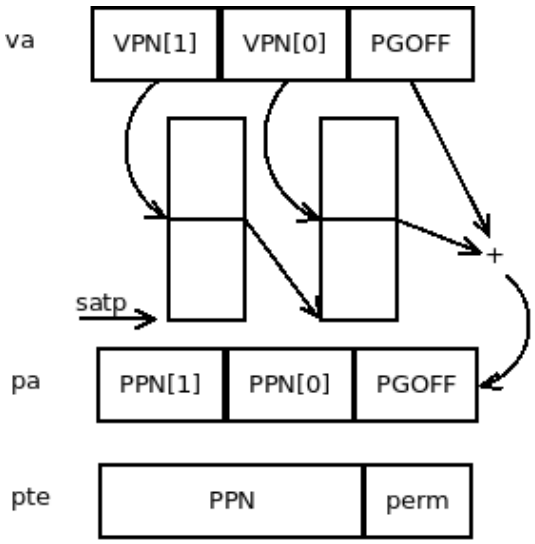


图 4-3 Sv32 的内存映射关系

2. 在 QEMU 中运行奔溃的原因分析。

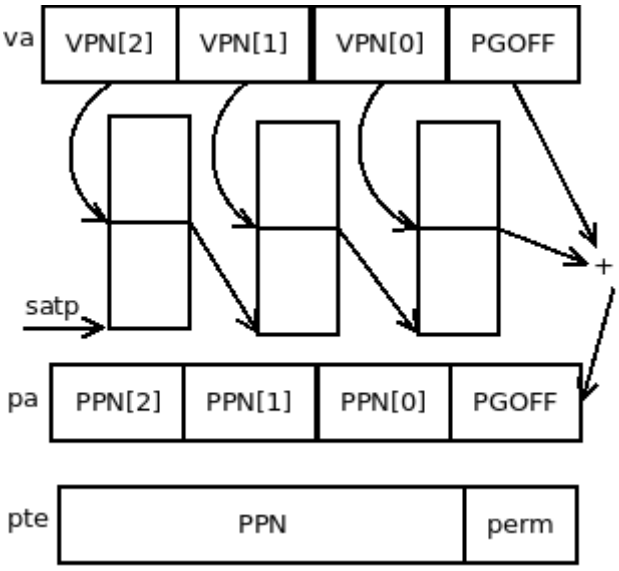


图 4-4 Sv39 的内存映射关系

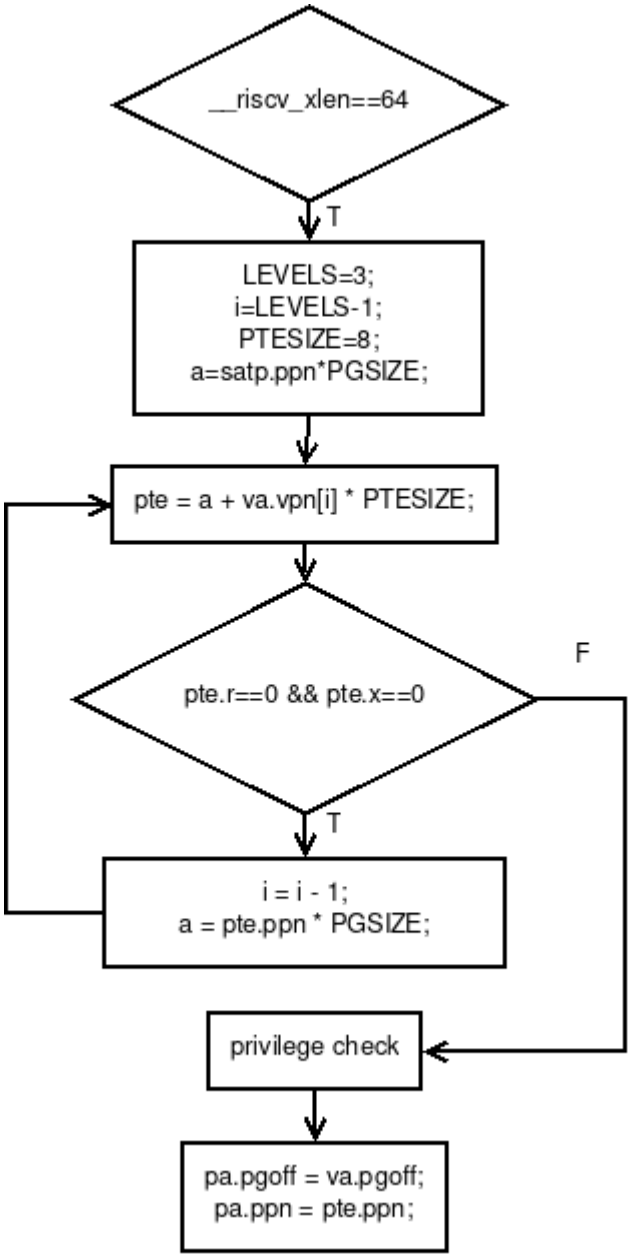
第一次运行的时候，奔溃在了 init_pmm_manager 函数，它的调用关系是 init_pmm_manager->pmm_init->kern_init。其实它奔溃很正常，因为 RV32 是二级页表映射，而 RV64 是三级或四级页表映射，映射关系是不一样的。只有深入两种映射关系之间的差异，才能进行正确的移植。

RV32 使用的是 Sv32 内存管理系统。

Sv32 将虚拟地址映射到物理地址的过程与 X86 是相同的，页表的基址放在 satp 寄存器内。所不同的是 Sv32 的虚拟地址是 32 位的，而物理地址是 34 位的。虚拟地址可以寻址 4G 的空间，而物理地址可寻址 16G 的空间。这样设计应该是为了支持虚拟化，即虚拟机上的不同系统可以使用不同的物理内存，互不干扰。

图 4-5 虚拟地址转换算法

RV64 可以使用的内存管理方案有 Sv39 和 Sv48 两种。Ucore 将使用 Sv39 内存管



理方案。Sv39 是基于页的 39 位虚拟内存系统。Sv39 支持 39 位的虚拟地址空间，其设计遵循 Sv32 的总体方案，仅仅是由 Sv32 的二级映射换到了三级映射。

Sv39 的虚拟地址是 64 位的，但只有低 39 位有效，且高 25 位必须和第 38 位相等（注：这里说的第 38 位指的是从 0 开始计数）。从虚拟地址到物理地址共需 3 级页表的转换。每一级的 PTE 都有可能是叶子 PTE，所以除了 4K 的页外，还有可能存在 2M 或 1G 的页，每种页都必须依照它自己的大小在虚拟层面和物理层面对齐。

Sv32，Sv39，Sv48 的转换算法都是一样的，不同的只有 LEVELS 和 PTESIZE 的值。LEVELS 指的是 vpn 有几级，PTESIZE 指的是 PTE 占几个字节。对于 Sv39 来说，由于是三级映射所以 LEVELS 取值为 3，每一个页表入口是 64 位的所以 PTESIZE 取值为 8。图 4-5 描述了在 Sv39 规约下，从虚拟地址 va 翻译到物理地址 pa 的过程。并且在图 4-5 里假设每个页的大小都占用 4k 字节，并没有其它大小的页，对于 Ucore 来说这样的假设是合适的。

3. 在 QEMU 中运行奔溃的问题处理。

需要为 Ucore 创建页表的三级映射以满足 Sv39 规约的要求。

在 kern/mm/mmu.h 里定义了与页表映射相关的宏和常量。

一个 Sv39 的虚拟地址是 64 位的，但只有低 39 位有效，低 39 位的有效地址可以分成 4 个部分。第 1 部分(0~11 位)是页内偏移地址，其所对应的宏是 PGOFF(la)，Sv32 和 Sv39 的页都是 4K 大小，所以宏 PGOFF(la)不需要修改。第 2 部分(12~20 位)是页在页表内的偏移量，其所对应的宏是 PTX(la)，由于页表索引的长度从 Sv32 的 10 位变成了 Sv39 的 9 位，所以对宏 PTX(la)需要进行修改，其运算的过程是虚拟地址右移 12 位然后取低 9 位的值，常量 PTXSHIFT 记录的就是 la 为计算页表索引所需移位的数值。第 3 部分(21~29 位)记录的是页表在次级页目录的偏移量，第 4 部分(30~38 位)记录的是次级页目录在顶级页目录的偏移量。对于 Sv32 来说只有一级页目录，所以只需一个宏 PDX 即可。但 Sv39 有二级页目录，需要两个宏 PDX1 和 PDX0 分别来代表顶级页目录和次级页目录，其运算过程是虚拟地址右移若干位然后取低 9 位的值，同时 PDX1 和 PDX0 所需移位的量也需要新建两个常量 PDX1SHIFT 和 PDX0SHIFT 来对应之。

Sv39 的页表条目是 64 位的，但只有低 54 位有效，低 54 位可以分成 2 个部分。第一部分(0~9 位)描述的是页的属性。第二部分(10~53 位)描述的是页的地址，使用宏 PPN(la)来表示，无需修改。

Sv39 的物理地址是 64 位的，但是只有低 56 位有效，低 56 位可以分成 2 部分。第一部分(0~11 位)来自于虚拟地址的 PGOFF。第二部分(12~55 位)来自于页表条目。

宏 PGADDR 的作用是把各级偏移量组装成一个虚拟地址。对于 Sv32 来说这样的偏移量有 3 个：d(页目录偏移量)，t(页表偏移量)，o(页内偏移量)。而 Sv39 则有 4 个偏移量，页目录的偏移量变成了两个：d1(顶级页目录偏移量)，d0(次级页目录偏移量)。所以，宏 PGADDR 需要依据 Sv39 的规约进行修改。

在 kern/mm/pmm.c 里定义了物理内存管理的具体实现，也要依据 Sv39 规约的要求进行修改。核心在于对 get_pte 函数的修改，get_pte 函数的作用是利用页目录来找到虚拟地址 la 所在页表的地址。Sv32 有二级页表，而 Sv39 有三级页表，get_pet 函数把对第二、三级页表的操作抽象成了对二级页表的操作，这样可以尽量复用代码，减小移植的工作量。

4. 在 QEMU 中运行奔溃的第二次修正。

经过如上修正，Ucore 在总体上应该是已经满足了 Sv39 规约的要求了。然而，当它在 QEMU 上运行的时候，依然出现了问题。会在某些特定的点奔溃，这与 Ucore 上物理内存管理的实现相关，要正确解决这些问题需要对物理内存管理的过程有一个清晰的认识。

物理内存管理的功能实现在 `pmm_init()` 函数。pmm_init 实现了两个功能，一是在

高层实现虚拟地址与物理地址的映射，二是在底层把空闲内存组织成一个链表进行管理。

要理解物理内存管理子系统，必须要理解两个重要的结构体：pmm_manager 和 Page。

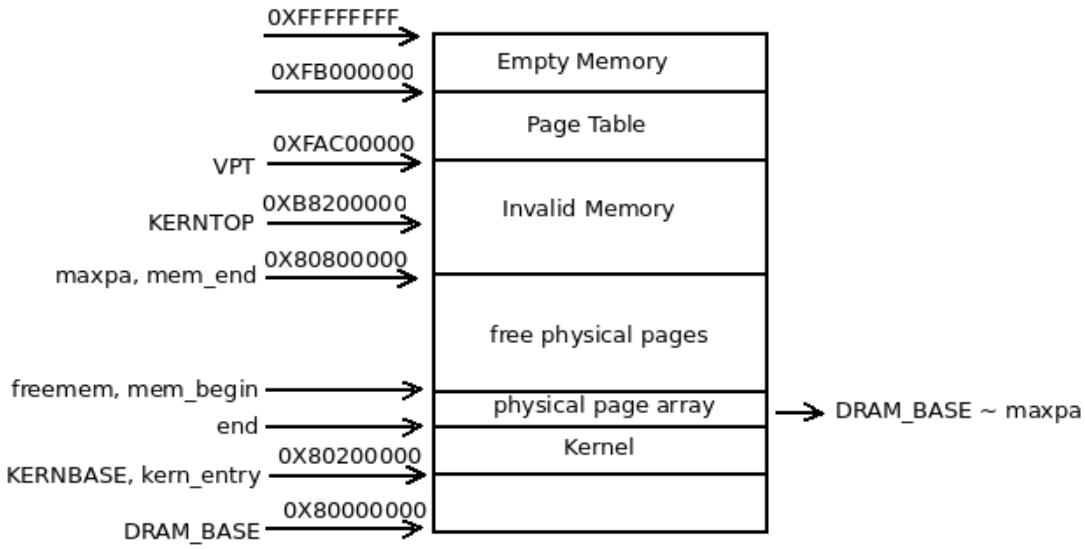


图 4-6 虚拟内存映射图

pmm_manager 结构体是物理内存管理的底层结构，通过它可以方便地将不同的物理内存管理算法实现到 Ucore 上。

```
struct pmm_manager {
    const char *name;                // pmm_manager 的名称
    void (*init)(void);              // 数据初始化
    void (*init_memmap)(struct Page *base，size_t n);    // 从 base 开始的连续 n
个页作为一个空闲块，按照地址从小到大的次序插入到链表 free_list 里
    struct Page *(*alloc_pages)(size_t n);    // 分配 n 个页
    void (*free_pages)(struct Page *base， size_t n);    // 从 base 开始释放 n 个页
    size_t (*nr_free_pages)(void);      // 返回空闲页的数量
    void (*check)(void);                // 正确性检验
};
```

Page 结构体是对物理页的描述，每一个物理页都对应着一个 Page 结构体。变量 flags 的 0 位是 PG_reserved，为 1 表示此页是由内核保留的，不可以分配和释放，为 0 表示它是普通的页，可以正常分配和释放。flags 的 1 位是 PG_property，为 1 表示此页是空闲块的第 1 个页且可分配，为 0 表示此页不是空闲块的第 1 个页或者虽然是空闲块的第 1 个页但已经被分配出去了。

```
struct Page {
    int ref;                // 有多少个虚拟页映射在此物理页上
    uint64_t flags;                 // 描述页帧的状态
    unsigned int property;          // 不同的算法有不同的意义，对于 first fit 算法来说，
它指空闲块的数量
    list_entry_t page_link;     // 链表 free_list 对此结构体的锚点
    list_entry_t pra_page_link;     // 用于页置换算法
    uintptr_t pra_vaddr;            // 用于页置换算法
};
```

通过对物理内存管理子系统的深入认识，就可以找到运行时奔溃的原因所在了。需要对系统做如下的修正。在 pmm.h 和 pmm.c 里使用的 sfence.vm 指令已经不再被支持了，依据 RISC-V 指令集手册所述，应使用改进的 sfence.vma 指令。satp 寄存器由 32 位变为了 64 位，相应位的意义也都不同了，需要进行相应的修改。再加上对一些数据类型的修改，加上了物理内存管理的 Ucore 可以在 QEMU 上正常运行了。

4.6 虚拟内存管理子系统的移植

虚拟内存管理子系统实际上是把硬盘空间作为内存的缓存。把暂时不用的内存页放到硬盘上，从而扩展了内存的大小。

1. 虚拟内存管理子系统运行时奔溃分析。

虚拟内存子系统的移植需要以物理内存管理子系统的修改为基础。当运行 make 命令进行编译时是可以通过编译的。据分析，这是由于虚拟内存管理子系统与物理内存管理子系统对底层接口的调用是完全一样。由于现有工具链的限制，使只能在内存中模拟硬盘所致。实际上虚拟内存管理子系统至少应增加对硬盘的控制接口。这一部分可以随着工具链的进一步完善，或者直接在开发板上支持 Ucore，而得以实现增加对

硬盘的管理接口。

然而当在 QEMU 中运行时，系统奔溃了。系统提示奔溃所在的位置是 kern/mm/vmm.c，奔溃的原因是一个叫 nr_free_pages 的变量和它之前保存的值不一致所致。为解决此问题，需要对虚拟内存管理子系统进行一些分析。

有三个重要的结构体是理解虚拟内存管理子系统的关键。

vma_struct 是虚拟内存区域(vma)的结构体。

mm_struct 描述进程在内存中分布的情况，是进程内存管理的核心数据结构。它控制的是具有相同 PDT 的 vma 的结构体。

swap_manager 结构体是虚拟内存管理子系统的底层结构，它是页置换算法的框架，可以方便地实现各种页置换算法。

结合如上分析，可以不致在虚拟内存管理子系统中迷路。通过进一步将奔溃时将不一致的两个变量的值打印出来，可以发现它们正好相差 1。经分析，相差的这个 1 是由于 Sv39 是三级页表映射，所以映射如果要建立的话会比 Sv32 多占用一个内存页。

2. 虚拟内存管理子系统的问题解决。

从根本上来说，物理内存管理子系统的 get_pte 函数决定了虚拟内存子系统此问题的出现，要彻底解决此问题还是要通过修改 get_pte 函数。目前解决此问题的方法，还是通过人为的修改 nr_free_pages 的值来实现的。

4.7 进程管理子系统的移植

由于进程的机制，操作系统才能进行多任务切换，它是现代操作系统的核心特征。所以，当操作系统维护好自己的跑马场后，紧接着就是要实现进程管理子系统的功能。

1. 以虚拟内存管理子系统的移植为基础。

首先需要修改 Makefile 文件使其使用 RV64 的工具链；其次修改 kernel.ld 使其将 kernel 的装载地址改为 0x80200000；然后修改 defs.h 创建新的数据类型；接下来修改 mmu.h 里宏的定义使其支持 Sv39 内存系统； 然后需要修改 pmm.c 文件使其支持支持三级页表，核心工作是修改 pmm.c 文件里的 get_pte 函数，在 get_pte 函数里用了偷懒的办法，就是将 Sv39 的二级映射和三级映射都封装成二级映射来处理，这样可以尽量复用以前的代码，减少移植的工作量；由于 sfence.vm 指令在 priv1.10 已不再支持，需要将所有 sfence.vm 指令修改为 sfence.vma 指令；所有与地址相关的变量长度均需改为 64 位；修改 memlayout.h 文件的 KERNBASE 和 struct Page；修改 vmm.c 的变量 nr_free_pages_store 以适应三级页表。

2. 进程管理子系统的运行。

进程管理子系统在运行的时候会触发 CAUSE_FETCH_PAGE_FAULT 异常。这一问题是由多个原因共同决定的，需要对进程管理子系统有一个深入的认识。

在底层接口方面，进程管理子系统在 process 文件夹下增加了 entry.S 和 switch.S 两个文件。user/libs/syscall.c 文件和 process/proc.c 文件里内嵌了汇编指令 。process/entry.S 的作用是跳转到新的内核线程中，process/switch.S 的作用是进行内

核线程间栈帧的切换。user/libs/syscall.c 里的汇编指令是用户态程序进行系统调用的代码，这将会触发一个异常进而执行内核态里的代码。process/proc.c 是里的汇编语言是内核态程序调用用户态程序的代码，它同样会触发一个异常进而执行用户态程序。

进程控制块是进程管理中的核心数据结构。

idle 是内核创建的第一个线程，它的作用是主管其它进程的调度。init_main 是第二个内核线程，它的作用是创建第三个内核线程 user_main 。user_main 是第三个内核线程，它的作用是创建用户进程，用户进程是不会返回的，如果返回了则说明内核存在错误，第二个内核线程 init_main 则会报告这个错误并退出系统。

do_execve 函数的作用是将用户程序载入内存从而创建一个新的用户进程。

do_exit 函数的作用是释放当前进程占用的除 CPU 外的其它资源，其后由父进程将当前进程释放。

do_fork 函数的作用则是为当前进程分配除 CPU 外的其它资源。

do_kill 函数的作用是在进程控制块里创建一个标志位，trap 函数会依据此标志位调用 do_exit 函数让此线程死掉。

do_wait 函数用于父进程回收子进程的资源。

do_yield 函数让 schedule 函数再重新调度一次。

3. 进程管理子系统的移植过程。

1) 内核线程看来与硬件无关，无须更多的更改即可运行。

2) syscall.c 里函数的参数实际上是放在 64 位的寄存器里，所以其对应的变量类型也要改为 64 位的。

3) kernel_execve 函数用于内核态线程切换到用户态进程，所以其使用的变量均需改为 64 位的，其使用的指令也需要改为对 64 位变量进行操作的指令。

4) riscv.h 里的 lcr3 函数用于在 satp 寄存器里保存一级页表的地址，也需要修改 stap 寄存里变量的存放格式符合 Sv39 的要求。

5) 由于在进程管理子系统里使用了用户态程序，那么决定用户态程序文件格式的 elf.h 文件，也需要修改为支持 elf64 的规约。

6) amo 指令操作 32 位的数据有可能会产生地址未对齐异常，在 defs.h 文件内需要把 `bool` 定义为 `long long` 类型。

4.8 调度子系统的移植

由于进程管理子系统本身就有一个简单的调度功能，而且调度子系统本身并没有使用更多的底层接口，所以其本身并没有需要修改的地方。当进程管理子系统的修改作用于调度子系统后，它就可以正常运行了。

4.9 进程间通信子系统的移植

进程间通信是通过同步或互斥来实现的。同步是要求进程间的通信是有一个先后的次序，互斥不要求进程间通信的次序，但同一时刻只能有一个进程访问资源。

1. 进程间通信子系统运行奔溃分析。

进程间通信子系统需要以进程管理子系统的修改为基础。然而当系统运行触发 static_assert 宏的时候会奔溃。

RV32 版本的进程间通信功能的实现从根本上说是依赖于原子操作，其使用的底层接口在 atomic.h 里定义。

在 init_main 函数里增加了 check_sync 函数，它是进程间通信子系统的主控函数。它完成了两个功能，一是用信号量解决哲学家问题，二是用管程解决哲学家问题。

2. 进程间通信子系统问题的解决。

通过对 kern/debug/assert.h 里 static_assert 进行分析，发现 static_assert 宏未实现，目前是直接注释掉了。当系统再次运行时，即可在 QEMU 上正常运行了。

4.10 文件系统的移植

计算机系统的所有资源都是以文件的形式呈现给用户的，同时文件系统也是操作系统里最复杂的子系统。总的来说，文件系统移植的特点就是重要性高，难度大。

1. 文件系统运行时的奔溃分析

文件系统的移植要以进程间通信子系统的修改为基础。

在移植的时候碰到的问题很奇怪，sh.c 里的 main 函数的参数 argc 为 1，但表达式(argc==1)的值为 0，表达式(argc>2)的值反倒为 1 了。要解决此问题需要对文件系统有深入的理解。

由于当前的硬盘是在内存里划了一块区域模拟的，所以文件系统并没有在进程间通信子系统的基础上使用其它的底层接口。但在真实的系统中，I/O 设备都是被封装成了文件系统的。

文件系统的作用是把块设备里的块抽象成文件。在文件系统的底层看到的是块，而在文件系统的上层看到的是文件。

从编译过程来看，`user` 目录下的 c 文件都编译到了 `disk0` 目录下，`disk0` 目录下的文件都放到了 sys.img 里，swap.img 里是一堆零，sys.img 和 swap.img 和 `kern` 目录下的文件都被链接到了 kernel 文件里，kernel 文件又被链接到 bbl 里，ucore.img 就是重命名后的 bbl。在这过程中 mksfs 起了重要的作用，感觉要好好看看它的原理。

从对底层接口调用的角度来看，`user` 目录下用到汇编的地方有两处处，一是 `libs/syscall.c` 文件切换到内核系统调用的部分，二是 libs/initcode.S 文件跳转到函数入口处的汇编指令。这两处用到汇编的地方均是用户空间与内核空间转换之处。其它地方 C 代码的指令由编译器保证符合 RV64，那么需要特别留意的就是这两处汇编的指令和传递的参数了。

`kern/fs` 目录存放的是文件系统的源码，感觉需要注意的是参数的长度，分清楚哪里用 64 位的参数，哪里用 32 位的参数。

2. 文件系统的问题解决。

在函数 load_icode 和_start 里要注意栈是 64 位的，要对相应的结构体和变量进行适当的调整。

4.11 Bug 分析与修复

问题描述：有的时候不能进入 trap 函数，需要在 trap 函数里加一条 cprintf 语句才能正常执行。

解决方案：在 trapentry.S 文件里加一条`.align 2`指令使所有的命令 4 字节对齐。

原因分析：存放中断服务例程(__alltraps)入口地址的寄存器 **stvec** 要求地址是 4 字节对齐的，当所有文件连接到一块的时候，trapentry.S 的汇编指令恰好在 trap 函数的后面。如果 trap 函数的末尾没有 4 字节对齐，则 trapentry.S 里的__alltraps 例程也就不能 4 字节对齐，这就导致中断发生的时候不能访问正确的中断地址。而在 trap 函数里加上 cprintf 函数后，trap 函数的末尾恰好 4 字节对齐了。

第五章 Ucore 上支持多处理器的设计

本章将介绍 XV6 上多处理器调度的作法[2]，以及本文对多处理器调度的设计。

5.1 XV6 对多处理机的支持

xv6 在启动主 cpu 和其他 cpu 时屏蔽中断。然后由每个处理器的调度器打开中断。为了控制一些特殊的代码片段不被中断，xv6 在进入这些代码片段之前关中断(例如 switchuvm)。

picirq.c 是早期主板的简单的可编程中断控制器(PIC)的管理代码。XV6 默认使用 SMP，所以 pic 是关闭的状态。

ioapic.c 处理的是多处理器 I/O 系统中的中断，它维护了一个表 (PRT，Programmable Redirection Table)，处理器通过 MMIO 读写这张表而实现中断管理。在 XV6 有一个全局变量 ioapic 就是这个可编程重定向表。Ioapicread 函数的作用是读取相应寄存器的内容，ioapicwrite 函数的作用是向相应寄存器写入数据，ioapicenable 函数的作用是为指定 CPU 使能中断。Ioapicinit 函数的作用则是对 ioapic 进行初始化设置且不开启任何中断。

lapic.c 是关联到每一个处理器上的中断。在 XV6 有一个全局变量的数组 lapic[]，每一个 CPU 都是通过读写这个数组实现对 lapic 的控制。Lapicw 函数是向数组 lapic[] 写入数据，lapicinit 函数是初始化 lapic 中断控制器[6]。

swtch.S 是上下文切换的代码。存在两种类型的切换：内核线程和调度器线程之间的切换，用户进程和内核线程之间的切换。

在 proc.h 里定义了上下文的结构 struct context，它决定了上下文切换的次序。

```
    struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

首先用%eax 保存 struct context **old 的指针，%edx 保存 struct context *new 的值，然后依次将%ebp， %ebx， %esi， %edi 入栈，然后将当前%esp 保存在*old 中，将%esp 切换到 new，最后将之前保存的寄存器的值出栈并返回[7]。

xv6 调用 swtch 函数的地方只有两处：一处是是在 proc.c 的 sched 函数里，从内核线程切换到某个 CPU 的调度器线程；一处是在 proc.c 的 scheduler 函数里，从某个 CPU 的调度器线程切换到内核线程。

proc.c 里的调度部分，是共行程序 sched 和 scheduler。

进程释放 CPU 的过程：获得进程表的锁 ptable.lock，释放拥有的其它锁，修改自己的状态 proc->state，调用 sched。yield， sleep 和 exit 都遵循了这个约定。

sched 首先检查进程此时是否持有锁，CPU 是否关闭了中断。然后调用 swtch 切换到调度器上下文，返回到 mycpu()->scheduler 中。

scheduler 则是一个双层 for 循环。在内层循环中找到一个可运行的进程(p->state==RUNNABLE)，然后将当前 cpu 的 c->proc 设为该进程，用 switchuvm 切换到该进程的页表，标记该进程为 RUNNING，再调用 swtch 切换到该进程中运行。在外层循环中要释放锁并显式地允许中断。因为如果一个闲置的调度器如果一直持有锁，其它 CPU 就不能进行调度操作了，如果不允许中断其它进程的 I/O 就永远无法到达了。

5.2 Linux 对 BBL 的支持

setup_arch：被外部的 start_kernel(init/main.c)调用。设置内存与多处理器[8][9]。

trap_init：被外部的 start_kernel（init/main.c)调用。被内部的 smp_callin(kernel/smpboot.c)调用。设置 sscratch 为 0，告诉异常处理程序我们在内核中；为 stvec 指定异常处理程序 handle_exception (kernel/entry.S)；设置 sie 寄存器为-1 使能所有的中断。

setup_smp：被 setup_arch 调用。设置多处理器。

__switch_to：通过被 switch_to(asm/switch_to.h)封装的形式被外部的 context_switch (kernel/sched/core.c)调用。

SAVE_ALL：宏，在进入系统调用或异常前将所有的寄存器保存在栈里。

RESTORE_ALL：宏，在从系统调用或异常返回前将栈中的值再保存到寄存器里。

handle_exception：入口，异常处理程序。

ret_form_fork：入口，从 fork 返回。

ret_from_kernel_thread：入口，从内核线程中返回。

__fstate_save：入口，文件状态保存。

__fstate_restore：入口，文件状态重新保存。

excep_vect_table：入口，异常向量表。

_start：是整个内核的入口。

对于 riscv 系统来说，Linux 是通过扁平设备树来管理硬件的。FDT 主要由三大部分组成：Header， Structure block， Strings block[10]。

Header 主要描述设备树的基本信息。

```
struct fdt_header {
  uint32_t magic;          // 魔数，固定为 0xd00dfeed
  uint32_t totalsize;      // 整个设备树的大小
  uint32_t off_dt_struct;  // Structure block 的偏移地址
  uint32_t off_dt_strings; // Strings block 的偏移地址
  uint32_t off_mem_rsvmap; // 内存保留区的偏移地址，该区不能被内核动态分配
```

```
  uint32_t version;         // 设备树版本
  uint32_t last_comp_version; // 向下兼容版本号
  uint32_t boot_cpuid_phys;     // 多处理器中用于启动的物理 CPU 的 ID
  uint32_t size_dt_strings;     // Strings block 的大小
  uint32_t size_dt_struct;      // Structure block 的大小
};
```

Structure block 以节点的形式保存目标板的设备信息，节点之间是树形结构，是设备树的主体。

```
#define FDT_BEGIN_NODE  1   // 一个 node 的开始位置
#define FDT_END_NODE    2   // 一个 node 的结束位置
#define FDT_PROP    3       // 一个 property 的开始位置
#define FDT_NOP     4       // 空节点
#define FDT_END     9       // Structure block 的结束位置
// 节点信息
struct fdt_scan_node {
  const struct fdt_scan_node *parent;   // 父节点的指针(树形结构)
  const char *name;     // 节点名
  int address_cells;    // 总线地址需要几个 cell(32bits 为 1 个 cell)，默认为 2
  int size_cells;       // 子总线地址需要几个 cell，默认为 1
};
// 节点属性
struct fdt_scan_prop {
  const struct fdt_scan_node *node;// 节点指针
  const char *name; // 属性名
  uint32_t *value;  // 属性值
  int len; // 属性值的长度，以字节为单位
};
```

一个节点的结构如下[11]：

- 节点开始标志：FDT_BEGIN_NODE
- 节点路径或节点单元名(version<3:节点路径；version>=16：节点单元名)
- 填充字段(对齐到 4 字节)
- 节点属性：FDT_PROP+属性值的字节长度(4Byte)+属性名的偏移地址(4Byte)+属性值和填充(对齐到 4 字节)
- 如果存在子节点，则定义之。
- 节点结构标志：FDT_END

Strings block 存放的是属性名，字符串块的引入避免了重复存储冗余的属性名，它的目的就是为了节省存储空间。

5.3 ucore 上对称多处理的设计思路

Ucore 计划在顶层实现 XV6 的多核调度，在底层则模仿 Linux 对 BBL 的调用。所以 Ucore 计划实现的调度算法是轮转法，仅仅是让每个进程轮流执行。

用户进程的切换过程如图 5-1 所示，用户进程先切换到内核线程，内核线程再切换到单个处理器的调度线程，每个处理器都有自己的调度线程，调度就发生在调度线程中。等调度完成后，再切换回内核线程，最后再切换到用户进程执行[12]。

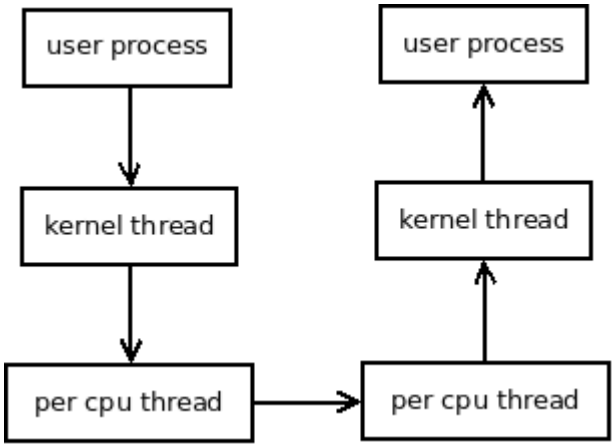


图 5-1 用户进程切换过程

对于处理器线程来说，最核心的是要获取处理器的 ID。Ucore 计划像 XV6 那样，将处理器的的信息都放到一个结构体中，并把所有处理器的结构体都以数组的形式放在内存中。

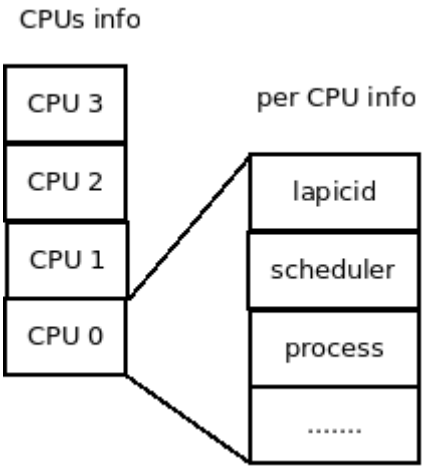


图 5-2 CPU 信息的存储

Ucore 线程切换的过程计划依照 XV6 的流程来做。首先是保存函数参数，然后将

旧线程的寄存器入栈，然后切换线程的栈，接下来将新线程的寄存器出栈，最后执行返回指令。由于栈和栈上的参数都被重新设置了，所以可以进入新线程中。可以发现线程的切换是轻量级的，比较节省系统的开销。

Ucore 主核激活其它核的过程则计划模仿 Linux。首先要屏蔽所有的中断。然后选择一个核来执行启动序列。接下来执行多核的设置，首先所有的内核线程都要共享同样的内存区域，然后将异常处理例程保存在相应的寄存器中，接下来初始化计数器，通知从核启动，将从核设置为在线，刷新 TLB，使能中断，最后设置多个 CPU 进入调度中。然后就进入内核的主控程序中，完成各个子系统的初始化工作。

可以发现，虽然这样的设计支持 SMP，但当一个核运行时其它核是处于无法获取锁的状态，从运行队列的角度来看只有一个 CPU，所以无需做什么更改即可使多 CPU 共享运行队列[13]。

5.4 Ucore 上不对称多处理的设计思路

不对称多处理的想法来源于中国传统思想，太级者无级而生，动静之机，阴阳之母也 。计算机上不对称调度生成的过程和阴阳两仪生成的过程应该是一致的。不对称多处理器想法的实例来源于阿里巴巴的混部技术。混部是把不同的运算集群混合部署在一起，两个互补的集群可以分担对方的压力，从而减少各自扩建的成本，减少浪费[14] 。如果把整个混部的集群看作一个计算机，那每一个数据中心都可以看作是一个核而这些核要完成的功能是各不相同的。所以异构调度的设计可以参考混部技术。

异构核是不对称多处理设计的核心。所谓异构的核，有两层意思。一是多个核的地位不一样，一个是主核其它是从核。二是每个核上要完成的任务不一样，比如可以分为 I/O 密集型的任务和计算密集型的任务。基于以上两个原因，导致每个核对资源占有的优先级也是不一样的。

主控核不是固定不变的，而是在多个核之间通过选举产生的。主核选举应在固定时间片内进行，且一定得有一个主核被选举出来。一般应选负载较低的核做为主核。这样可以比较容易的实现系统不重启更新。

主核负责给所有的核分配适合其执行的进程(包括主核自己)。

由于从核对资源的权限不同，导致适合其完成的任务不同。从核负责完成主核交付的进程，并对执行的进程画像。

在应用软件的层面，可以采用给软件画像的方法分别软件的特点，从而为进程在核之间的分配提供依据。进程画像是对进程特征的描述，它是保存在进程控制块里的一个 n 元组，n 元组里的每一项都表明进程的一个特征。

第六章 实验验证

6.1 中断子系统的验证

当中断子系统正确设置好后，中断处理例程会捕获计时器中断，以大概每秒钟一次的频率在屏幕上输出少量字符串。当屏幕上可以稳定连续地输出字符串时，即表明中断子系统已成功移植。

6.2 内存管理子系统的验证

对于物理内存管理子系统，Ucore 会使用三个函数进行验证：check_alloc_page, check_pgdir, check_boot_pgdir。check_alloc_page 是对物理内存管理子系统底层功能的检验，它会从各种角度分配和释放内存空间，并在分配和释放的过程中间进行记数，以验证内存管理算法的正确性。check_pgdir 是对内存映射正确性的检验，它从建立和取消各种内存映射的角度来进行验证。check_boot_pgdir 也是对内存映射正确性的检验但它会在虚拟内存中写入数据，通过对比虚拟内存中的数据是否和其对应的物理内存中的数据是否一致来进行验证。

通过以上三个函数的验证，可以基本确定物理内存管理子系统不存在大的错误了。当屏幕上连续输出中断子系统所设定的字符串时，即表明物理内存管理子系统的移植已成功完成。

对于虚拟内存管理子系统，Ucore 使用了两个函数进行验证：check_vma_struct 和 check_pgfault。check_vma_struct 函数是对虚拟内存区域进行验证，它通过对虚拟内存区域进行创建和删除操作以判断虚拟内存区域的映射是否正确。check_pgfault 是对缺页处理例程进行正确性的判断，它通过手工设置的方式使内核发生缺页置换，进而验证正确的内存页能否再被挽回到内存上。

通过 check_vam_struct 和 check_pgfault 两个函数的验证，可以判断虚拟内存管理子系统可以正确运行了。最后，依然是会在屏幕上连续输出少量字符串。

6.3 进程管理子系统的验证

对于内核线程管理子系统来说，只是创建了两个内核线程。当调度发生的时候，第二个线程能正确创建，并在屏幕上打印出线程的一些信息，即表示内核线程子系统成功运行了。

内核线程子系统成功运行的时候，会在屏幕上打印内核线程的一些基本信息并打印“Hello World”，然后直接退出系统。

对于用户进程管理子系统来说，是创建了几个用户进程。当调度发生的时候，会进行用户进程的创建和销毁，用户进程会输出相关信息。当在屏幕上看到用户进程输出的相关信息时，即表明用户进程子系统移植成功了。

6.4 调度子系统的验证

在进程管理子系统里已经有一个简单的调度子系统了，本实验只是实现了一些其它的调度算法。所以，当进程可以成功调度的时候，即表明本系统已成功运行。

调度子系统和进程管理子系统的屏幕输出是完全一样的。

6.5 进程间通信子系统的验证

进程间通信子系统是在内核线程里加了一个 check_sync 函数来实现对其验证的 。check_sync 函数通过让内核实现的进程间通信机制来解决哲学家问题验证其正确性。

屏幕上会打印出哲学家问题的解决过程，通过屏幕输出来验证进程间通信子系统是否成功建立。

6.6 文件系统的验证

Ucore 在文件系统上实现了包括 shell 在内的若干用户态程序。如果文件系统被正确创建并挂载，则 shell 及其它用户态程序可以正确执行。

对文件系统验证是通过看 shell 及其它用户态程序是否可以正确加载并执行来间接判断的。

第七章 总结与展望

移植 Ucore 和设计 Uocre 支持多处理器的过程，加深了我对操作系统和底层硬件的理解。同时，对于以前在其它课程中学习了但不理解或模糊的地方，也随着对底层的深入理解而清晰起来。然而，对于操作系统这样的复杂系统，我依然如海边的小孩一样无知，深感有大量需要学习和探索的领域。如下是将它们列举出来的尝试，挂一漏万，仅仅是一个微小的预测。

从内存管理子系统的角度来看，Ucore 能控制的内存空间实际只有 8M，未来可以尝试实现对全部内存空间的控制。Ucore 交换分区目前是做在内存中的，未来可以尝试将其实现在硬盘中。内存管理子系统还带有很大的 x86 的特色，还可以针对 riscv 做一些改进。Sv32 和 Sv39 实现逻辑非常相似，可以尝试一套代码同时支持 Sv32 和 Sv39。

从文件系统的角度来看，Ucore 的文件系统目前是在内存中的，未来可以将其实现在硬盘上。

随着摩尔定律逐渐走向瓶颈，必须依靠多处理器协同才能得到更强大的运算能力。所以在 Ucore 上实现支持多处理器是大有益处的。实现像 XV6 那样的 SMP 是一个比较容易的开端。然而，SMP 的弊端是随着核数增加到一定程度，系统的效率会逐渐降低。不对称多处理的设计可以解决以上问题，相信未来会有一个比较好的发展。

未来的一个趋势应该是软硬件协同设计。既然可以在操作系统层面实现不对称多处理，那么也可以尝试实现异构核的开发板，支持异构调度的操作系统。

从方便教学的角度来看，BBL 还是太过于庞大和复杂了，不方便学生聚集于操作系统。BBL 是为像 Linux 那样的大型系统软件设计的，Ucore 其实用不到 BBL 的大部分功能。未来可以不用 BBL，而是专为 Ucore 设计一个精简的 Boot Loader。

从指令集来看，RV64I 和 RV32I 高度相似。从内存管理规范来看，Sv32 和 Sv39 的处理逻辑也是高度相似的。从 BBL 层开始就已经弱化 RV64 和 RV32 之间的差异，向操作系统层提供统一接口了。所以，应该进一步修改 Ucore，使它可以用一套代码同时支持 32 位和 64 位两个架构。

参考文献

[1] 陈渝,向勇. ucore_os_docs [EB/OL]. https://github.com/chyyuu/simple_os_book, 2018.

[2] Russ Cox, Frans Kaashoek, Robert Morris. xv6 — a simple, Unix-like teaching operating system [EB/OL]. http://pdos.csail.mit.edu/6.828/xv6/, 2012-08-28.

[3] 包云岗. RISC-V 前沿研究与思考[EB/OL]. 中科院计算所, 2018.

[4] Andrew Waterman, Krste Asanovi´c, SiFive Inc., CS Division, EECS Department, University of California, Berkeley. riscv-spec-v2.2 [EB/OL]. https://riscv.org/specifications/, 2017.

[5] Andrew Waterman, Krste Asanovi´c, SiFive Inc., CS Division, EECS Department, University of California, Berkeley. riscv-privileged-v1.10 [EB/OL]. https://riscv.org/specifications/, 2017.

[6] 百度. XV6 操作系统整体报告[EB/OL]. https://wenku.baidu.com/view/339ba16e7e21af45b307a8e6, 2012.

[7] Intel inc.. 80386 manual [EB/OL]. https://pdos.csail.mit.edu/6.828/2014/readings/i386/toc.htm, 2018.

[8] 金刚, 吴军, 马鹏, 任敏华. 嵌入式 Linux 操作系统移植中 SMP 的实现研究[J]. 信息技术, 2016: 93-96.

[9] 朱苏建, 邵培南, 金刚. 嵌入式 Linux 在 SMP 系统上的移植研究与实现[J]. 电子设计工程, 2016: 93-100.

[10] 蔡人和. 基于 ARM Cortex-A9 MPCore 嵌入式多核操作系统内核研究与实现[D]. 成都：电子科技大学, 2016.

[11] 郝玉胜. uC/OS-II 嵌入式操作系统内核移植研究及其实现[D]. 兰州：兰州交通大学, 2014.

[12] 郭海林. 基于多核 DSP 的嵌入式操作系统 RTEMS 的移植研究[D]. 合肥：中国科学技术大学, 2015.

[13] 蔡玉鑫. 嵌入式多核处理器核间通信方法的设计与实现[D]. 西安：西安电子科技大学, 2015.

[14] 阿里巴巴. 混部技术[EB/OL]. http://36kr.com/p/5119365.html, 2017.