

清 华 大 学

综 合 论 文 训 练

题目：软硬协同的用户态中断机制研究

系 别：电子工程系

专 业：电子信息科学与技术

姓 名：尤予阳

指 导 教 师：马洪兵 教授

联合指导教师：陈 渝 副教授

2022 年 6 月 21 日

中文摘要

现代处理器通常具有多特权级架构设计，并限制中断仅能由高特权级程序处理。传统的驱动设计需要将访问硬件、处理中断的部分置于高特权级的内核部分，而低特权级的用户程序通过系统调用等方式间接访问外设硬件资源。随着近年来网络、存储等 I/O 设备的通信带宽迅速发展，跨越特权级边界的切换开销逐渐成为应用的性能瓶颈，因此工业界和学术界研究出若干绕过内核直接访问外设的方案，但这类方案受前述特权级架构限制，无法使用中断机制，通常只能采用轮询方式访问外设和协作式任务调度，这可能导致较高的处理器占用率和负载不均衡等问题。

本文从软硬件协同优化的角度入手，设计了用户态中断的硬件行为规范和系统软件适配方案，并开发了原型平台和基于用户态中断的硬件驱动程序。具体工作可分为以下部分：

- 对 RISC-V 的用户态中断扩展草案进行完善，规定用户态中断所需的寄存器和指令行为；将现有 PLIC 设计中的上下文概念扩展到用户态，以支持将外设产生的中断导入用户态。
- 在 QEMU 模拟器和 FPGA 平台上开发了硬件原型，用于快速验证规范设计方案及软件平台的实现正确性，估计用户态中断在真实硬件处理器上的性能表现。
- 基于 rCore 操作系统实现对用户态中断的支持，允许多个并行运行的用户程序互不干扰地使用用户态中断机制。
- 开发基于传统中断、内核态的和基于用户态中断和轮询的、运行在用户态的外设驱动，在硬件原型平台上测试其在吞吐量、延迟、处理器占用方面的性能表现，验证系统设计的有效性。

关键词：用户态中断；操作系统；设备驱动

ABSTRACT

Modern CPU typically adopts multi-privilege level architectures and restrict interrupts to be handled only by more privileged programs. Traditional driver designs require that access to hardware and handling of interrupts be placed in the OS kernel (which runs at higher privilege), while user programs with lower privilege level access peripherals indirectly through system calls and other means. With the rapid development of bandwidth of network, storage and other I/O devices in recent years, the switching overhead across the privilege level boundary has gradually become a bottleneck for application performances, so industry and academia have researched several solutions that access to peripheral directly, bypassing kernel. But such solutions are limited by the aforementioned privilege level architecture thus cannot use interrupts, and usually only use polling to access peripherals and cooperative task scheduling. This can lead to problems such as higher processor utilization and load imbalance.

In this paper, we design the hardware behavior specification and system software adaptation scheme for user-state interrupts from the perspective of software and hardware co-optimization, and develop a prototype platform and hardware drivers based on user-mode interrupts. The specific work can be demonstrated as the following parts.

- Refining RISC-V's user-mode interrupt extensions and specifying the register and instruction behavior required for user-mode interrupts; extending the context concept from existing PLIC designs to user mode to directing interrupts raised from device into user space.
- Developing hardware prototypes on the QEMU simulator and FPGA platform for agile verification of the correctness of the specification design and software platform implementation, and for estimating the performance of user-mode interrupts on real hardware processors.
- Implementing support for user-mode interrupts based on the rCore operating system, allowing multiple user programs running in parallel and utilizing the user-mode interrupt mechanism non-interferingly.
- Developing traditional interrupt-based, kernel-mode, and user-mode interrupt- and polling-based device drivers running in the user space, and test their performance

in terms of throughput, latency, and processor utilization on a hardware prototype platform, verified the effectiveness of the system design.

Keywords: user-mode interrupt; operating system; device driver

目 录

第 1 章 引言	1
1.1 研究背景	1
1.1.1 处理器的特权级	2
1.1.2 RISC-V 指令集的中断（陷入）机制	2
1.1.3 特权级切换的性能开销	3
1.2 相关工作	3
1.2.1 RISC-V 指令集的用户态中断扩展 (N 扩展)	3
1.2.2 x86 指令集的用户态中断机制 (UINTR)	4
1.3 论文结构安排	11
第 2 章 系统设计	12
2.1 寄存器和指令行为规范	12
2.1.1 用户状态寄存器 (ustatus)	12
2.1.2 用户陷入向量基址寄存器 (utvec)	13
2.1.3 用户中断寄存器 (uip 与 uie)	13
2.1.4 内核态陷入委托寄存器 (sedeleg 与 sideleg)	14
2.1.5 用户态暂存寄存器 (uscratch)	15
2.1.6 用户异常程序计数器 (uepc)	15
2.1.7 用户陷入原因寄存器 (ucause)	15
2.1.8 用户陷入值寄存器 (utval)	15
2.1.9 用户态中断返回指令 (URET)	17
2.2 用户态中断的处理过程	17
2.2.1 中断的产生	17
2.2.2 异常的产生	18
2.2.3 中断和异常的处理	18
2.3 外部中断与平台级中断控制器 (PLIC)	18
2.3.1 PLIC 上下文	20
2.3.2 中断领取与完成	20
2.3.3 PLIC 的用户态中断扩展	21

2.4 操作系统内核对用户态中断的管理	21
2.4.1 用户态中断与其他特权级中断的对比	21
2.4.2 用户态中断上下文	22
2.4.3 外设中断控制	22
2.4.4 中断转发和注入	24
2.5 应用程序接口	26
2.5.1 系统调用	26
2.5.2 用户态中断处理函数	27
2.5.3 用户程序示例模板	28
2.6 设计比较	29
第 3 章 系统实现	31
3.1 模拟器与 FPGA 原型平台	31
3.2 启动器	32
3.3 操作系统内核	33
3.3.1 多核启动流程	33
3.3.2 进程调度	33
第 4 章 性能测试与分析	35
4.1 测试环境	35
4.2 驱动吞吐率测试	35
4.3 驱动延时比较	36
4.3.1 延时测量方法	36
4.3.2 内核驱动延时	36
4.3.3 用户态中断驱动延时	38
第 5 章 总结与展望	41
5.1 本文内容总结	41
5.2 未来工作展望	41
插图索引	42
表格索引	43
参考文献	44
致 谢	47

附录 A 外文资料的书面翻译.....	48
---------------------	----

主要符号表

DPDK	数据平面开发套件 (Data Plane Development Kit)
SPDK	存储性能开发套件 (Storage Performance Development Kit)
FPGA	现场可编程逻辑门阵列 (Field Programmable Gate Array)
KPTI	内核页表隔离 (Kernel Pagetable Isolation)
IPC	跨进程通信 (Inter-Process Communication)
UPID	用户态中断发布描述符 (User Posted Interrupt Descriptor)
UITT	用户态中断目标表 (User Interrupt Target Table)
HART	硬件线程 (Hardware Thread)
ABI	应用程序二进制接口 (Application Binary Interface)
SBI	监管者二进制接口 (Supervisor Binary Interface)
PLIC	平台级中断控制器 (Platform Level Interrupt Controller)
MMIO	内存映射的输入/输出 (Memory-Mapped Input/Output)
CLINT	核心本地中断器 (Core-Local Interruptor)
ACLINT	高级核心本地中断器 (Advanced Core-Local Interruptor)

第 1 章 引言

1.1 研究背景

现代应用处理器通常采用多特权级的设计，限制低特权级的程序访问一些特殊寄存器、内存区域或执行特权指令，以提高系统的安全性与隔离性。另一方面，中断机制允许处理器暂停当前正在运行的代码，转而处理某些突发事件，这样可以让处理器及时响应外部事件，同时不必浪费时钟周期来不断轮询外部世界的状态。中断通常由在高特权级下运行的软件（如操作系统内核）处理，而低特权级的用户程序无法感知到中断的存在。但特权级的切换也有一定的性能开销，随着网络、存储等 I/O 设备的性能提升，这种开销逐渐成为系统 I/O 的性能瓶颈，由内核处理外设中断，再将数据传递给应用程序的硬件驱动模型难以满足需求。

为消除这一瓶颈，研究者们提出了用户态 I/O 的概念。例如在 Linux 操作系统中，用户可以将物理外设绑定到特殊的内核驱动——`uio` 或 `vfiio` 上，此时外设的地址空间、PCI、DMA 等资源将直接映射给用户空间，如此便在数据通路上去除了内核，消除了特权级切换以及内核中其他组件对性能的影响。高性能网络常用的数据平面开发套件 (Data Plane Development Kit, DPDK)^[1] 和高性能存储常用的存储性能开发套件 (Storage Performance Development Kit, SPDK)^[2] 就采用了这种方案，允许用户程序直接访问网卡和非易失存储。但现有的处理器指令集架构大多不允许用户程序访问中断，这些方案只能采用轮询方式访问外设以及协作式任务调度。轮询的优势在于具有较好的局域性，对缓存友好，劣势在于处理器占用率高，一旦发生任务切换性能就会受到较大影响，在实际部署中往往需要让操作系统内核让对应的用户程序独占一部分处理器核；而 Kaffes 等人在 2019 年的一项研究^[3] 中指出，协作式任务调度策略可能在负载呈现长尾或多模态分布时，出现较为严重的负载不均衡，尾延迟大幅提升。

本项目提出了一种用户态中断机制，允许用户程序直接处理中断，从而消除了应用访问外设时的特权级切换开销，提升驱动程序的性能同时可以降低处理器占用，在 FPGA 平台上实现了具有用户态中断机制的处理器原型，在 rCore 操作系统中实现了对用户态中断的管理机制，以及使用用户态中断机制的驱动程序，测试了用户态中断带来的驱动性能提升。

1.1.1 处理器的特权级

现代应用处理器通常采用多特权级的设计，限制低特权级的程序访问一些特殊寄存器、内存区域或执行特权指令，以提高系统的安全性与隔离性。如 x86 架构有 Ring0 到 Ring3^[4]，Arm-A 指令集划分了 EL0 到 EL3^[5]，而 RISC-V 指令集规范^{[6]1-4}将 CPU 的运行状态划分为机器 (Machine), 监管者 (Supervisor), 用户 (User) 这三个特权级。M 态为最高的特权级，也是所有 RISC-V 平台必须实现的特权级，M 态下的指令对 CPU 有完全的控制权；S 态为通常的操作系统运行的特权级，U 态则为用户程序运行的特权级。

在实现了地址空间隔离的平台上，M 态程序不受地址空间限制，始终使用物理地址访问内存，除非将 `mstatus.MPRV` 置位；S 态程序可以通过 `satp` 寄存器控制分页模式和页表基址，通常在初始化完成后运行在虚拟地址上；U 态程序则完全运行在虚拟地址空间中，受页表权限控制位的限制，且无权访问和修改 `satp` 寄存器的内容。

在引入了虚拟化、可信执行等机制的处理器上，特权级设计会更加复杂，如 x86 架构的虚拟机扩展 (Virtual Machine Extension, VMX) 进一步划分了 Root 和 Non-Root 模式^[7]，RISC-V 则加入了 HS, VS, VU 特权级^{[6]99}，本文不会重点讨论这些内容。

1.1.2 RISC-V 指令集的中断（陷入）机制

RISC-V 将陷入 (trap) 分为同步的异常 (exception) 和异步的中断 (interrupt)，异常由指令执行产生，而中断通常来源于指令之外的因素，如时钟、外设等。中断分为外部中断、时钟中断、软件中断，分别记为 `xEI`, `xTI`, `xSI` (`x` 为特权级，下同)；这三者在每个特权级下有不同的中断编号，因而 M 态的时钟中断和 S 态的时钟中断不是同一个中断。与之相对的是，异常在不同特权级下编号相同，因为异常源于指令执行，而执行某条指令时系统的特权级是确定的，无需通过编号区分。

中断的使能和屏蔽由 `xstatus.xIE` 位和 `xie` 寄存器控制，前者为当前特权级下的全局中断使能，后者可以独立控制每种中断的使能情况。待处理中断在 `xip` 寄存器中相应的位为 1。低特权级的 `status`, `ie`, `ip` 寄存器均为相应高特权级寄存器的子集，即高特权级可以查看和修改低特权级的中断控制信息，但反过来不行，如 S 态和 M 态的程序均可以通过清除 `sie.STIE` 来屏蔽 S 态的时钟中断，但 S 态程序不能通过访问或修改 `mie.MTIE` 位来影响 M 态时钟中断的行为。这也是特权级架构提供的隔离性的体现。

默认情况下所有的陷入都由最高特权级（即 M 态）的程序来处理，高特权级的程序可以再将其转交给低特权级处理，如将 `xtval`, `xepc`, `xcause` 写入相应的低特权级寄存器，清除 `xip` 寄存器中的相应位，再将低特权级的中断处理函数入口地址（即 `ytvec` 寄存器的值）移入 `xepc` 中，执行 `xret` 指令，即可进入到低特权级的中断处理程序。

为了提高中断处理效率，RISC-V 提供了陷入委托机制，允许陷入在较低特权级处理，相应的控制寄存器为 `xideleg` 和 `xedeleg`。高特权级的陷入不能委托给低特权级。对于已经委托的低特权级陷入，在高特权级下将被忽略，直至返回低特权级时才进行处理。如 `mideleg` 中 `MTI`, `MEI`, `MSI` 的相应位恒为 0，不能委托，而将 `mideleg.STI` 置为 1 则将 S 态的时钟中断委托给 S 态处理，此时即使 `mip.STIP==1`，M 态也不会进入中断处理程序。

1.1.3 特权级切换的性能开销

在应用程序发起系统调用、出现运行错误或时间片到期等情况下，需要切换到高特权级的内核进行处理，处理完成后内核再返回（可能是另一程序的）用户空间，这一过程中就发生了特权级切换。在切换特权级时，通常需要完成保存和恢复通用寄存器、切换栈指针、修改某些控制寄存器、跳转到特定的处理函数等一系列动作，在开启了内核地址空间隔离 (Kernel Pagetable Isolation, KPTI) 特性的内核中，还需要切换页表，这些是特权级切换的直接开销；而处理器微架构层面的缓存和 TLB 未命中、分支预测失效等带来的性能下降称为特权级切换的间接开销。Zeyu Mi 等人在 2019 年对跨进程通信 (Inter-Process Communication, IPC) 的一项研究表明，直接开销大约为两千个 CPU 周期，而间接开销约为三千个 CPU 周期^[8]。

1.2 相关工作

1.2.1 RISC-V 指令集的用户态中断扩展 (N 扩展)

在 RISC-V 特权级指令规范 v1.12 的草案中存在一章用户态中断扩展（即 N 扩展^①）的草案^[9]。该扩展设计的主要目标是为嵌入式系统提供更好的安全支持，此类系统可以仅实现 M 和 U 两个特权级，将“不可信”的用户程序代码置于 U 态执行，同时通过中断委托机制，允许其直接处理中断。在类 Unix 系统上，该扩展主要希望为整数溢出、浮点异常、垃圾回收等发生在用户态的事件提供支持。

^① RISC-V 指令集的标准扩展常以一个字母命名，如整数乘除法扩展为 M，单精度浮点为 F，原子指令扩展为 A，虚拟化扩展为 H 等。

在社区讨论中有一些观点，认为在嵌入式系统上，使用 M 和 S 两个特权级可以实现同样的目标，只需将 satp 寄存器由硬件置为 0；而对于 Unix 环境下，N 扩展的应用场景尚不明朗，长期以来也几乎无人推动 N 扩展的完善和实现。^[10] 最终在 RISC-V 规范 v1.12 版本中，N 扩展被移除，其他部分关于用户态中断的表述也相应降级为“未来可能的扩展”。

目前已知实现 N 扩展的处理器核有赛昉科技的昉·天枢^[11]、晶心科技的 AX25MP, AX27, AX45^[12]、印度理工学院的 SHAKTI-C^[13]，其中前四者为闭源的商用 IP 核，SHAKTI-C 则开源了其 RTL 代码。

1.2.2 x86 指令集的用户态中断机制 (UINTR)

Intel 在 2021 年 5 月发布的 Intel 指令集架构拓展^[14] 中加入了 x86 平台上的用户态中断规范，在 2022 H2 发布的 Sapphire Rapids 系列处理器中提供了硬件实现。在硬件架构上，中断接收方是用户空间的任务，而发送方可以是另一个用户任务、内核或外部设备。目前 Intel 在 Linux 内核中实现了用户态任务之间的中断机制，微测试表明，基于 UINTR 实现的 IPC 机制相较于纯软件的信号、eventfd、管道等，在延迟方面有约一个数量级的性能提升^[15]。

x86 中的用户态中断被定义为架构中的新事件，可以在 CPL=3 的 64 位模式下传递给软件进行处理，不需要改变段状态。不同的用户中断通过一个 6 位的用户中断向量区分，在传递中断时被压到栈上。新增一个 UIRET（用户中断返回）指令用于退出中断处理上下文。

用户中断架构由新的内核管理的状态进行配置，这个状态包括新的 MSR，在内核切换线程时进行更新。其中一个 MSR 指向名为用户态中断发布描述符 (User Posted Interrupt Descriptor, UPID) 的数据结构，用户态中断可以发布到与某个线程关联的 UPID 中。在接收到一个普通中断后，处理器将根据 UPID 中的内容将其识别为用户态中断并传递给软件，这一过程称为用户中断通知。

系统软件可以定义用于发布用户中断和发送用户中断通知的操作。用户中断架构定义了一条新指令 SENDUIPI，应用软件可以使用这条指令发送处理器间用户中断（用户态 IPI）。执行 SENDUIPI 指令时，处理器会根据用户中断目标表 (User Interrupt Target Table, UIITT) 的内容，在相应 UPID 中发布一个用户中断，并发送一个用户中断通知。

UINTR 新增的处理器状态如表 1.1 所示

表 1.1 UINTR 相关状态定义

缩写	全称	描述
UIRR	Uintr Request Register	该状态为一个位向量，某位为 1 表示对应下标的用户态中断待处理
UIF	Uintr Flag	控制用户态中断是否使能
UIHANDLER	Uintr Handler	用户态中断处理函数入口地址
UISTACKADJUST	Uintr Stack Adjustment	中断处理栈的基址或偏移量
UINV	Uintr Notification Vector	识别为用户态中断处理的中断编号
UPIDADDR	UPID Address	UPID 的内存地址
UITTADDR	UITT Address	UITT 的内存地址
UITTSZ	UITT Size	UITT 中的表项数量

1.2.2.1 UINTR 的传递与处理

当 $UIRR \neq 0$ 时表示有待处理的用户态中断，任何更新 UIRR 的行为都会触发处理器的 UINTR 识别流程。当满足以下条件时，处理器就会将中断传递到指令边界上：

- $UIF = 1$ ，即 UINTR 已使能
- 没有阻塞中的 MOC SS 或 POP SS 操作
- $CPL = 3$ ，即处理器运行在用户态
- $IA32_EFER.LMA = CS.L = 1$ ，即处理器处于 64 位模式
- 软件没有运行在 Enclave 中

用户态中断传递的优先级仅低于普通中断，它会将处理器从 TPAUSE 和 UMWAIT 指令的状态中唤醒，但不会唤醒处于关闭或等待 SIPI 状态的处理器。用户态中断不会改变 CPL。如果 $UISTACKADJUST[0] = 0$ ，中断传递流程会将 RSP 减去 UISTACKADJUST，否则会将 RSP 替换为 UISTACKADJUST。传递流程总是将 RSP 对齐到 16 字节边界。用户态中断传递对栈的访问可能触发异常，在处理这些异常之前 RSP 会恢复到先前的值，如果这些异常产生了一个使用 EXT 位的错误码，该位会被清除。产生这种异常时，UIRR 不会更新，UIF 不会清除，在异常处理完成之后，处理器会继续识别和传递先前的用户态中断。用户态中断的处理流程如图 1.1 所示。

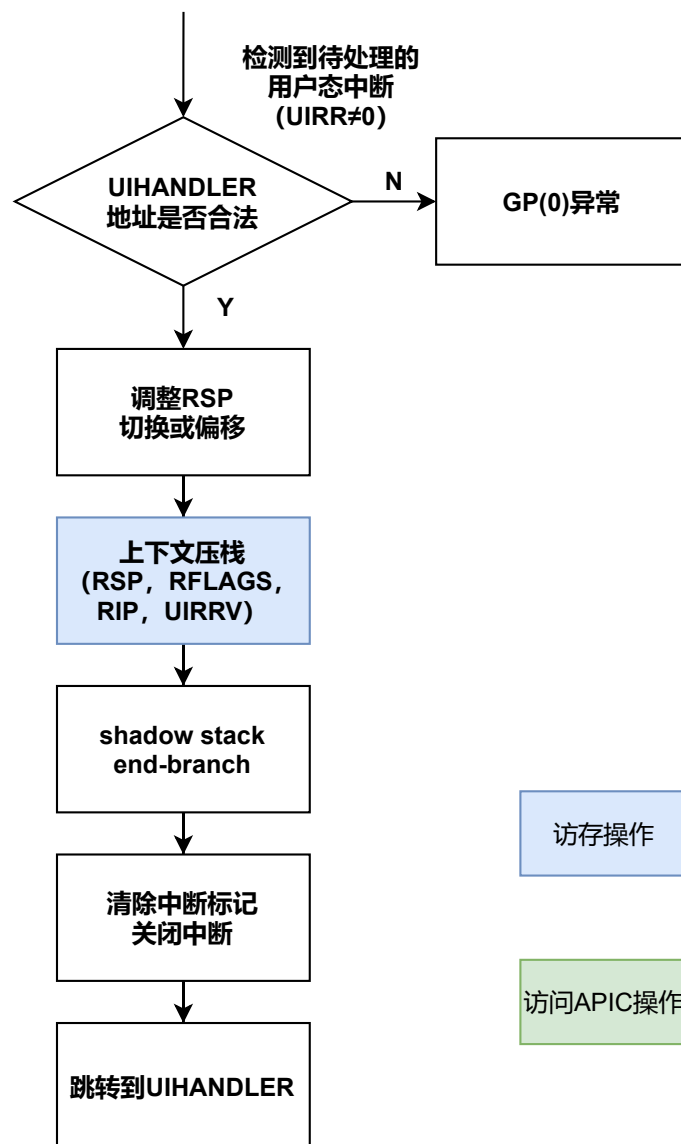


图 1.1 UINTR 的处理流程

1.2.2.2 UINTR 通知

用户态中断发布是平台主体或 CPU 上运行的软件将用户态中断记录到内存中的**用户态中断发布描述符 (UPID)**的过程。平台主体或软件可以向中断目标正在运行的逻辑处理器发送一个普通中断，该中断称**作用户态中断通知**。UPID 的结构定义如表 1.2 所示。

表 1.2 UPID 定义

位域	名称	描述
0	待处理通知 (ON)	若该位被置位表示 PIR 中有一个或多个待处理的 UINTR 通知
1	抑制通知 (SN)	该位被置位时，发布到该描述符的中断不产生 UINTR 通知
15:2	预留	通知过程会忽略这些位，SENDUIPI 要求这些位为 0
23:16	通知向量	由 SENDUIPI 使用
31:24	预留	通知过程会忽略这些位，SENDUIPI 要求这些位为 0
63:32	通知目标 (NDST)	目标的物理 APIC ID，由 SENDUIPI 使用。在 xAPIC 模式下，第 47:40 位为 8 位 APIC ID；在 x2APIC 模式，该域为 32 位 APIC ID
127:64	发布的中断请求 (PIR)	该段为一个位向量，如果某位为 1 表示有相应编号的用户态中断请求

当逻辑处理器收到一个普通中断时，首先会识别该中断是否为 UINTR 通知。识别过程如下：

1. 从局域 APIC 获取中断向量编号 V ；
2. 若 $V = UINV$ ，则该中断是 UINTR 通知，进行下一步；否则该中断为普通中断，通过中断描述符 (IDT) 进行处理，识别过程结束；
3. 处理器向局域 APIC 的中断结束 (EOI) 寄存器写入 0，这会将局域 APIC 中编号为 V 的中断清除，然后进入 UINTR 通知的处理流程。

UINTR 通知的处理流程（注意与 UINTR 本身的处理流程相区分）如下：

1. 逻辑处理器通过原子操作清除 UPID 中的待处理通知位；
2. 处理器将 UPID 中的 PIR 域读出，同时将其全部写为 0，该过程同样是原子的；

3. 处理器将 PIR 中为 1 的位在 UIRR 中对应置为 1，然后进入 UINTR 的识别流程。

上述流程是不可中断的，且第 1 步和第 2 步的访存在内核特权级下完成。

UINTR 通知的识别和处理流程如图 1.2 所示。

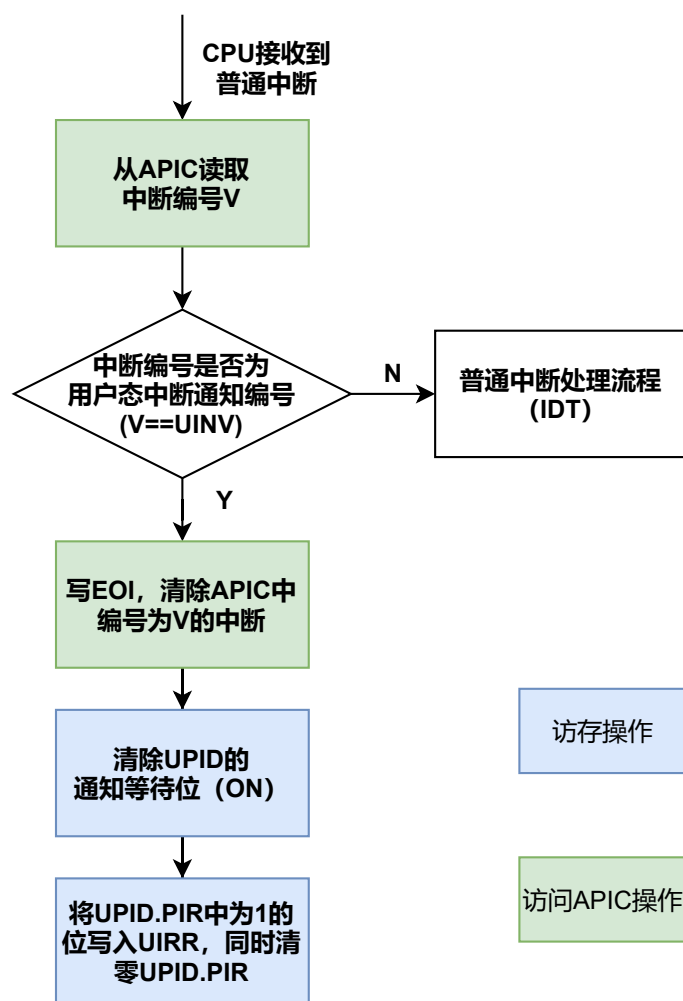


图 1.2 UINTR 通知的识别与处理

1.2.2.3 UINTR 的发送

软件可以通过 SENDUIPI 指令发送跨核的用户态中断。该指令接受一个寄存器作为操作数，寄存器中为 UITT 表项 (UITTE) 的下标。每条 UITTE 占 16 个字节，格式如表 1.3 所示。

SENDUIPI 指令首先会进行一系列合法性检查，如下标是否在 UITTSZ 范围内、UITTE 是否有效、UITTE 和目标 UPID 的保留位是否为 0 等。然后会根据目标 UPID 的 ON 和 SN 位判断是否发布 UINTR 通知，并将目标 UPID 中 PIR 对应

表 1.3 UITTE 定义

位域	名称	描述
0	V	有效位
7:1	预留	必须为 0
15:8	UV	发送的用户态中断向量编号
63:16	预留	
127:64	UPIDADDR	目标 UPID 的线性地址，对齐到 64 字节边界

位置 1。若需要发布 UINTR 通知，则根据目标 UPID 中的 NDST 和 UITTE 中的 UV，通过 APIC 发布一个跨核中断。SENDUIPI 指令的流程如图 1.3 所示。

1.2.2.4 UINTR 在内核中的管理

Intel 在 Linux 内核中实现了对 UINTR 的支持^[16]。发布和接收用户态中断的对象均为线程，每个线程的控制结构体中额外加入一个 `uintr_receiver` 和 `uintr_sender` 结构，前者为接收方维护 UPID，后者为发送方维护 UITT。此外还在内核中加入了一些辅助数据结构，如 `uintr_upid_ctx` 和 `uintr_receiver_info` 等，用于记录 UPID 和线程的绑定关系等。

UINTR 的接收和发送方都需要通过系统调用向内核进行注册。接收方注册时可以获得一个 `uintr_fd`，而发送方则需要获取该文件描述符，注册时即获得一个 `uipi_index`，用作 SENDUIPI 指令的参数。

在内核进行线程切换时，将被换出的任务的 `UPID.SN` 置位来禁用后续的 UINTR 通知，将被换入的任务的 `UPID.NDST` 更新，并清除 `UPID.SN` 启用通知，同时检查是否有待处理的用户态中断，如有则给本核发送一个 IPI 来产生 UINTR 通知。

内核还允许用户程序通过系统调用显式地等待用户态中断，实现方法为更改对应线程 UPID 的通知向量，然后将线程挂起，这样当其他线程给该线程发送跨核中断时，该中断会被识别为普通中断而转入内核处理，内核此时可以唤起之前被挂起的线程。

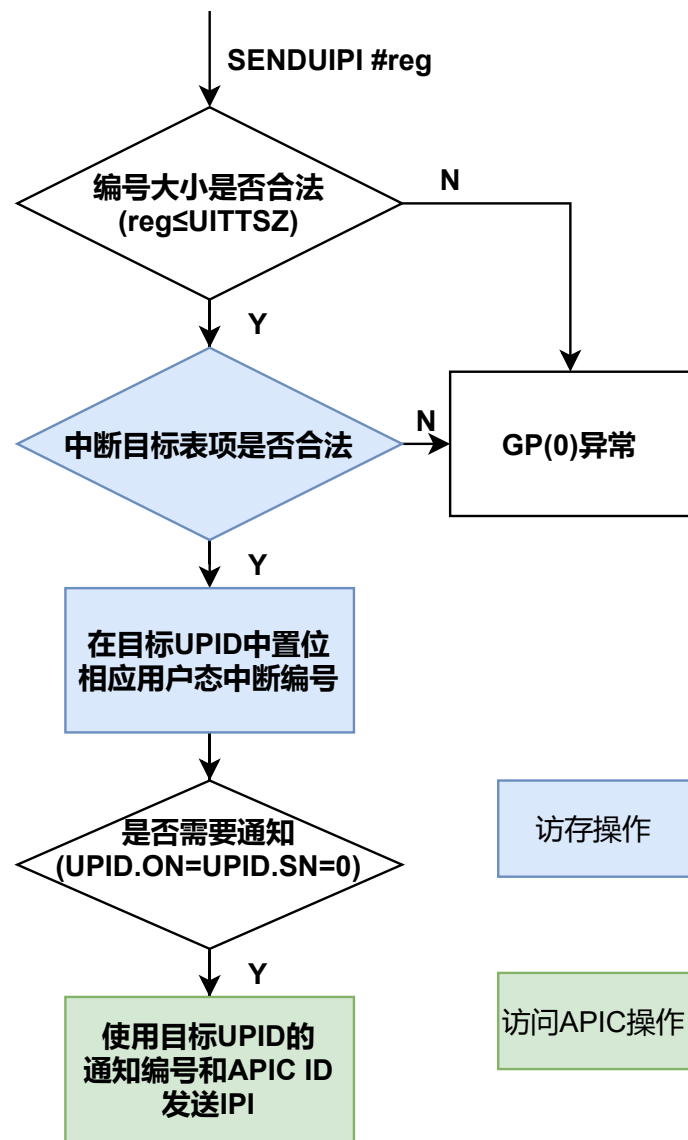


图 1.3 SENDUIPI 流程

1.3 论文结构安排

本文第一章介绍了研究的背景及其他相关工作的现状，概述了本研究的工作和文章结构；第二章阐述了本研究的整体设计，包括寄存器和指令规范、中断控制器、内核行为以及用户程序；第三章介绍了本研究的具体实施方案，原型平台的搭建和系统的初始化过程；第四章展示了研究的测试和实验成果，证明了本文设计方案带来的驱动性能提升；第五章对工作进行总结，对于设计和实现中的一些不足之处，展望了后续工作可能的优化方向。

第 2 章 系统设计

2.1 寄存器和指令行为规范

本项目最初基于 RISC-V 的 N 扩展设计，该扩展定义了一些用户态中断所需的寄存器和指令，但该扩展仅为草案，有较多章节尚未完成，且之后被移出指令集规范。本项目将该草案补充完全，作为后续工作的基础。

2.1.1 用户状态寄存器 (ustatus)

表 2.1 ustatus 寄存器定义

位域	位宽	字段
UXLEN-1:5 ¹	UXLEN-5	WPRI ²
4	1	UPIE
3:1	3	WPRI
0	1	UIE

¹ UXLEN 指用户态的控制与状态寄存器 (CSR) 位宽，在本工作中通常取值为 64

² WPRI 指该字段可能为未来预留，软件对该字段应采取“写入保留，读取忽略” (Reserved Writes Preserve Values, Reads Ignore Values) 的处理方式以确保前向兼容性，这是 RISC-V 特权级指令规范中的一种通用约定。

ustatus 是一个 UXLEN 位长的可读写寄存器，记录和控制硬件线程 (Hardware Thread, HART) 当前的工作状态。其格式定义如表 2.1 所示。用户态中断使能位 UIE 为零时，用户态中断被禁用。为了向用户态陷入处理程序提供原子性，UIE 中的值在用户态中断发生时被复制到 UPIE，且 UIE 被置为零。

UIE 和 UPIE 是 mstatus 和 sstatus 寄存器中对应位的镜像。

进入用户态中断处理函数之前的特权级只可能是用户态，所以不需要 UPP 位。

指令 URET 用于从用户态陷入状态中返回。URET 将 UPIE 复制回 UIE，然后将 UPIE 置位，最后将 uepc 拷贝至 pc。在 UPIE/UIE 栈弹出后置位 UPIE 是为了启用中断，以及帮助发现代码中的错误。

表 2.2 utvec 寄存器定义

位域	位宽	字段
UXLEN-1:2	UXLEN-2	BASE
1:0	2	MODE

表 2.3 utvec MODE 字段定义

值	名称	描述
0	直接	出现任何陷入时 pc 寄存器都会被设为 BASE
1	向量	$BASE + 4 * cause$
≥ 2	-	保留

2.1.2 用户陷入向量基址寄存器 (utvec)

utvec 是 UXLEN 位宽的可读写寄存器，存储陷入向量的设置，包括向量基址 (BASE) 和向量模式 (MODE) 字段。其格式定义如表 2.2 所示，模式字段定义见表 2.3。

utvec 中的 BASE 为 WARL^① 字段，可以存储任何有效的虚拟地址或物理地址，地址需要对齐到 4 字节。其他的向量模式可能有额外的对齐约束条件。

2.1.3 用户中断寄存器 (uip 与 uie)

uip 和 uie 均为 UXLEN 位的可读写寄存器，其中 uip 存储等待处理的中断信息，uie 存储相应的中断使能位。二者字段定义如表 2.4 所示

RISC-V 特权级架构定义了三种中断：软件中断、时钟中断和外部中断。用户态软件中断通过置位当前硬件线程的 uip 的软件中断等待位 (USIP) 来触发。清零该位可以清除待处理的软件中断。当 uie 中的 USIE 为零时，用户态软件中断被禁用。

ABI 应当提供一种向其他硬件线程发送跨核中断的机制，这最终将置位接收方硬件线程 uip 寄存器的 USIP 位。

除了 USIP，uip 中的其他位在用户态是只读的。

如果 uip 寄存器中的 UTIP 位被置位，将产生一个待处理的用户态时钟中断。当 uie 寄存器中的 UTIE 位被置零时，用户态时钟中断被禁用。ABI 应该提供清

^① WARL 指软件可以“写入任意值，读出合法值” (Write Any Values, Reads Legal Values)。与 WPRI 类似，这也是 RISC-V 特权级规范的对 CSR 的一种约定。

表 2.4 uip 与 uie 寄存器定义

位域	位宽	uip 字段	uie 字段
UXLEN-1:9	UXLEN-9	WPRI	WPRI
8	1	UEIP	UEIE
7:5	3	WPRI	WPRI
4	1	UTIP	UTIE
3:1	3	WPRI	WPRI
0	1	USIP	USIE

除待处理的时钟中断的机制。

如果 uip 寄存器中的 UEIP 位被置位，将产生一个待处理的用户态外部中断。当 uie 寄存器中的 UEIE 位被置位时，用户态外部中断被禁用。ABI 应该提供屏蔽、解除屏蔽和查询外部中断原因的机制。

uip 和 uie 寄存器是 mip 和 mie 寄存器的子集。对 uip/uie 任何字段的读取或写入操作，都会等效为对 mip/mie 的相应字段的读取或写入。如果系统实现了 S 模式，uip 和 uie 寄存器也是 sip 和 mie 寄存器的子集。

2.1.4 内核态陷入委托寄存器 (sedeleg 与 sideleg)

为提升中断和异常的处理性能，可以实现独立的可读写寄存器 sedeleg 和 sideleg，设置其中的位将特定的中断和异常交由用户态陷入处理程序处理。这两个寄存器与相应的机器态陷入委托寄存器 (medeleg 和 mideleg) 布局相同。只有已经被委托给 S 态的陷入对应的位才是可写的，其余位由硬件保持为 0，即只有委托给 S 态的陷入才可能被委托给 U 态。

当一个陷入被委托给一个权限较低的模式 u 时，ucause 寄存器被写入陷阱的原因；uepc 寄存器被写入发生陷阱的指令的虚拟地址；utval 寄存器被写入一个特定的异常数据；mstatus 的 UPIE 字段被写入陷阱发生时 UIE 字段的值；mstatus 的 UIE 字段被清零。mcause/scause 和 mepc/sepc 寄存器以及 mstatus 的 MPP 和 MPIE 字段不被写入。

一个合法的实现不应硬性规定任何委托位为一，也就是说，任何可以被委托的陷入都必须支持不被委托。一个实现方案是选择可委托的陷入的子集。支持的可委托位可通过向每个比特位置写 1，然后读回 medeleg / sedeleg 或 mideleg/sideleg 中的值，查看哪些位上有 1。

不会在用户态发生的的陷入对应位应由硬件保持恒为零, 如 ECall from S/H/M-mode

2.1.5 用户态暂存寄存器 (uscratch)

uscratch 寄存器是一个 UXLEN 位用户态可访问的读/写寄存器。通常该寄存器可用来保存一个指针, 该指针指向一个硬件线程独占的上下文结构, 并且在进入用户态陷入处理函数时由软件交换到通用寄存器中。

2.1.6 用户异常程序计数器 (uepc)

uepc 是 UXLEN 位的可读写寄存器。最低位 (uepc[0]) 恒为零。次低位 (uepc[1]) 视实现的对齐需求而定。

uepc 是 WARL 寄存器, 必须能存储所有有效的虚拟地址, 但不需要能够存储所有可能的无效地址。实现可以先将一些非法地址映射为其他非法地址再写入 uepc。

当陷入在用户态处理时, 被中断或触发异常的指令的虚拟地址被写入 uepc, 除此之外 uepc 永远不会被硬件实现写入, 但可能被软件显式写入。

2.1.7 用户陷入原因寄存器 (ucause)

表 2.5 ucause 寄存器定义

位域	位宽	字段
UXLEN-1	1	是否为中断
UXLEN-2:0	UXLEN-1	异常编号 (WLRL) ¹

¹ WLRL 指该字段应“读写均仅为合法值” (Write/Read Only Legal Values)。

ucause 是 UXLEN 位长读写寄存器, 其格式定义如表 2.5 所示, 各陷入的编码见表 2.6。当陷入在用户态处理时, 触发陷入的事件编号被写入 ucause, 除此之外 ucause 永远不会被硬件实现写入, 但可能被软件显式写入。

2.1.8 用户陷入值寄存器 (utval)

utval 是 UXLEN 位的可读写寄存器。当陷入在用户态处理时, 和特定异常相关的信息将被写入 utval 以帮助软件处理陷入, 除此之外 utval 永远不会被硬件实现写入, 但可能被软件显式写入。硬件平台指定哪些异常必须将信息写入 utval, 以及哪些异常会无条件写入 0。

表 2.6 ucause 在陷入中断之后的值

是否为中断	异常编号	描述
1	0	用户态软件中断
1	1-3	预留
1	4	用户态时钟中断
1	5-7	预留
1	8	用户态外部中断
1	9-15	预留
1	≥16	由平台使用
0	0	指令地址未对齐
0	1	指令访问错误
0	2	非法指令
0	3	断点
0	4	加载地址未对齐
0	5	加载访问错误
0	6	存储/原子内存操作地址未对齐
0	7	存储/原子内存操作访问错误
0	8	用户态环境调用
0	9-11	预留
0	12	指令页错误
0	13	加载页错误
0	14	预留
0	15	存储/内存原子操作页错误
0	16-23	预留
0	24-31	自定义用途
0	32-47	预留
0	48-63	自定义用途
0	≥64	预留

当硬件断点被触发，或是一个指令/加载/存储地址未对齐/访问错误/页错误异常产生时，导致错误的虚拟地址被写入 `utval`。当非法指令异常产生时，相应指令的前 `XLEN` 或 `ILEN` 位可能被写入 `utval`。对于其他异常，`utval` 被置为 0，但未来的标准可能重新定义 `utval` 的设置。

2.1.9 用户态中断返回指令 (URET)

表 2.7 URET 指令格式

位域	位宽	字段	取值
31:20	12	funct12	URET
19:15	5	rs1	0
14:12	3	funct3	PRIV
11:7	5	rd	0
6:0	7	opcode	SYSTEM

URET 将 `pc` 设置为 `uepc`，将 `ustatus.UIE` 设置为 `ustatus.UPIE`，从而恢复中断前的状态。其指令编码如表 2.7 所示。

2.2 用户态中断的处理过程

2.2.1 中断的产生

与 M 态和 S 态类似，中断分为软件中断 (Software Interrupt)、时钟中断 (Timer Interrupt) 和外部中断 (External Interrupt)。方便起见，三类中断简记为 `xSI`、`xTI`、`xEI`，其中 `x` 为特权级。

硬件或软件将 `uip.UXIP` (`X` 表示中断种类) 置为 1，硬件检测发现 `uip` 非零，进入中断的判断流程。首先检查该中断是否被委托给用户态处理，即 `sideleg` 寄存器中对应的位是否为 1；如果为真，检查用户态全局中断使能是否为真，即 `ustatus.UIE` 是否为 1；若仍为真，再检查该中断是否被使能，即 `uie.UXIE` 是否为 1；如果还为真，则进入用户态中断处理的流程。

需注意的是，上述寄存器中，`uie uip` 为 `mie mip` 的子集，即读写会同时作用于所有的 `xip xie`。而 `ustatus` 的 `UIE` 和 `UPIE` 与 `mstatus` 中的相同位相同。虽然上述中断产生的流程中有判断次序，实际实现中一般使用组合逻辑，将寄存器值进行位与来判断，可以认为是同时判断的。

2.2.2 异常的产生

当异常发生时，硬件只检查 `sedeleg` 寄存器中对应的位是否为 1，若为真则进入用户态异常的处理流程。

在 RISC-V 中，中断和异常的处理流程是统一的，下面出于描述简单考虑，多数情况下围绕中断进行描述。

2.2.3 中断和异常的处理

在上述的产生流程后，处理器开始进行一些预处理：

- 设置 `ustatus.UPIE` 为 `ustatus.UIE` 的值，并置 `ustatus.UIE` 为 0
- 根据中断来源设置 `ucause`
- 设置 `uepc` 为发生中断或异常时的 `pc`
- 根据中断来源设置 `utval`
- 根据 `utvec` 的最低二位和高位的基地址，跳转到设置好的中断处理程序

中断处理程序通常需要保存和恢复的现场上下文包括：

- `x1-x31` 通用寄存器 (如果确定中断处理程序中不会使用到某些寄存器，可以省去保存和恢复)
- `ustatus` (可能需要修改 `ustatus` 以运行嵌套中断)
- `uepc` (可能需要通过修改进一步触发 S 态的中断/异常)

2.3 外部中断与平台级中断控制器 (PLIC)

外部中断主要用于处理与外设相关的中断。RISC-V 架构的处理器广泛使用平台级中断控制器 (Platform Level Interrupt Controller, PLIC) 来复用数量较少的 `xEIP` 信号，允许处理器识别和处理数量更多的外设中断。PLIC 接收所有外设的中断信号，根据程序配置的优先级、阈值和上下文规则，在相应的硬件线程上触发外部中断。程序需要从 PLIC 的 MMIO 寄存器中进一步读取具体的中断外设源信息。

`mip.MEIP` 位对程序是只读的，只能由 PLIC 写入或清除。`mip.SEIP` 可读写，M 态程序可以将该位置 1，而是否产生 S 态外部中断由该位的值和 PLIC 相应的信号逻辑或的结果决定，二者中任一为 1 即产生中断；`csrr`、`csrrs` 和 `csrrc` 指令在该位上的行为略有不同，具体定义可见 RISC-V 特权级规范对应章节^{[6]33}。`sip.SEIP` (对 S 态程序) 是只读的。对于用户态中断的 `xip.UEIP` 位可以设计类似的约束条件。

目前的 PLIC 规范^[17] 支持至多 1024 个 (外设) 中断源和 15872 套中断上下

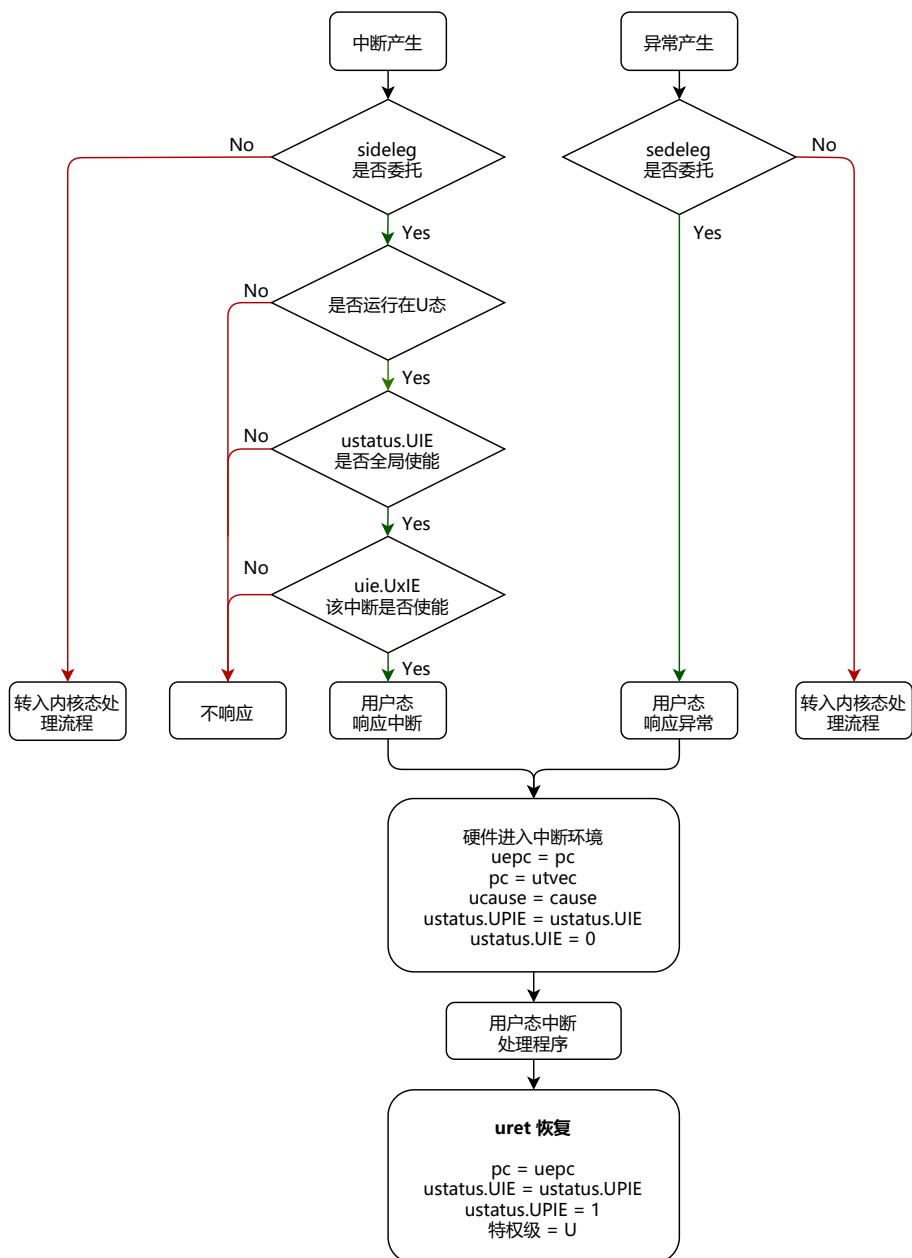


图 2.1 用户态中断的处理流程

文，每个中断源至多可配置 2^{32} 种优先级，每套上下文至多可配置 2^{32} 种优先级阈值。在每套上下文中，程序可独立配置每个中断源是否使能。

PLIC 的中断触发条件如下：

- 中断源产生中断等待信号；
- 在某套上下文中，该中断源被使能；
- 该中断源的优先级高于该上下文的优先级阈值；

上述条件均满足时，PLIC 会在硬件线程中触发外部中断，硬件线程编号与中断的特权级由上下文的设计决定。

2.3.1 PLIC 上下文

上下文指的是特定处理器中，特定硬件线程上的特定特权级，如果 CPU 中有三个硬件线程和两个可以处理中断的特权级（M 和 S），那么就存在六套 PLIC 上下文（在某个硬件线程上触发某个特权级的外部中断）。在该场景下，可以按表 2.8 方式对上下文进行编码。

表 2.8 uip 与 uie 寄存器定义

硬件线程	特权级	上下文编号
0	M	0
0	S	1
1	M	2
1	S	3
2	M	4
2	S	5

设某个中断源符合上下文 0、1 和 5 的中断触发条件，那么 PLIC 会在硬件线程 0 上同时将 MEIP 和 SEIP 置为有效，在硬件线程 2 上将 SEIP 置为有效；在硬件线程 1 上，该中断源不会触发外部中断。

2.3.2 中断领取与完成

PLIC 中每套上下文具有一个领取 / 完成 (claim/complete) 寄存器。程序读取该寄存器时，PLIC 会返回该上下文中优先级最高、等待信号有效且被使能的中断源编号（该中断源的优先级可以低于上下文阈值），清除该中断源的等待位，并全局屏蔽该中断源的中断有效信号。程序向该寄存器中写入一个中断编号以通知 PLIC

该中断处理完成，若相应中断源在其上下文中被使能，则 PLIC 解除相应的屏蔽，否则忽略本次写入。

PLIC 对于领取/完成寄存器的读写是“无记忆”的，写入的中断编号与领取的编号可以不同。如果一个中断源在两个上下文中被使能，可以在第一个上下文领取该中断并屏蔽其信号，在第二个上下文中完成该中断并解除屏蔽。

2.3.3 PLIC 的用户态中断扩展

PLIC 的设计基本能够与 N 扩展兼容，只需为每个硬件线程的 U 态额外分配一套上下文，并将 PLIC 相应的中断信号输出与 xip.UAIP 位相连即可。

每个上下文的领取 / 完成寄存器地址均与 4KB 边界对齐，故内核可以将 U 态对应上下文的地址直接映射到用户进程地址空间，这样用户进程可以直接完成领取 / 完成操作；中断源优先级和使能位对应的地址空间应当仅由内核访问和控制。

下一节将介绍如何在操作系统中实现对外部中断的管理和复用，使其在多进程环境下仍然能够正常运行。

2.4 操作系统内核对用户态中断的管理

2.4.1 用户态中断与其他特权级中断的对比

RISC-V 特权级指令架构规范中已经规定了机器态 (M) 和内核态 (S) 的中断规范，以及二者之间的互动机制，如特权级屏蔽、中断委托等。设计用户态 (U) 中断时，我们在一定程度上参照了现有的 S 态中断机制，并将 M 和 S 之间的关系平移到了 S 和 U 上，以保持整个中断架构的一致性。

在同时实现了 M、S 和 U，且没有实现 H 扩展的系统上，在 M 态运行的通常只有一个启动器（或者称为 SEE、SBI），且只在启动、发生 SBI 调用和处理部分中断时才执行代码；在 S 态运行的只有一个内核，一部分系统服务的代码可能在处理器上执行较长时间；这两个特权级只需要各自有一个陷入处理函数即可。

而在 U 态运行的通常有多个的用户程序，以时间片的方式轮流占用处理器执行。对于同步异常而言，这通常不构成问题，因为同步异常一定发生在某个程序执行自己的指令时触发；但对于异步中断，其来源往往无法知晓用户进程的状态，后者可能正在运行、处在调度队列中、进入睡眠甚至已经结束，如何确保进程收到自己想要的中断，同时不会错误地收到本应由其他进程处理的中断，就成了使用用户态中断时面临的核心问题。

2.4.2 用户态中断上下文

为了解决上述问题，我们提出了“用户态中断上下文”的概念。用户态中断上下文包括各中断寄存器、外部中断映射和待处理中断记录。进程切换时，内核保存当前进程的用户态中断上下文，恢复下一进程的上下文，从而确保在多核、多进程环境下，进程的中断执行流仍然可以正常运行。

2.4.2.1 中断记录

在 RISC-V 规范中，中断的原因存储于 `xcause` 寄存器中。但在多进程环境下，我们还需要对中断源进行区分，如发出信号的源进程、外部中断对应的外设编号等。我们在内核中加入了 `UserTrapRecord` 结构体实现这一目标：

```
#[repr(C)]
#[derive(Copy, Clone)]
pub struct UserTrapRecord {
    pub cause: usize,
    pub message: usize,
}
```

`cause` 中存储的内容基本与 `xcause` 寄存器保持一致，如对于 UTI 有 `cause=4`，对于 UEI 有 `cause=8`。对于 USI，由于信号本身的值需要填入 `message`，我们将源进程的 PID 编入 `cause` 中。标准的 `xcause` 编号为 0-15，只需 4 位，故我们令 `cause=PID<<4`。

2.4.2.2 中断缓冲区

当用户进程被调度离开硬件线程时，仍然有可能产生该进程需要处理的中断，此时由内核将中断记录暂存入该进程的中断缓冲区，待该进程再次被调度运行时进行处理。为了减少数据复制，我们将中断缓冲区设计为一个由进程和内核共享的完整的内存页，大小为 4KB，在内核中仅记录该页的物理页号，而在用户地址空间中该页有固定的虚拟地址。

在这一页上我们构造了一个无锁的、单消费单生产者的环形队列，内核和用户程序使用相同的数据结构定义，这样无需担心在用户程序处理中断时内核再次访问该缓冲区可能造成的数据竞争问题。

2.4.3 外设中断控制

内核需要记录用户进程申请了哪些外设中断，以及相应设备在 PLIC 中是否启用，以便在进程调度切换时进行配置。rCore-N 中进程切换基本流程如下：

1. 暂停当前进程，将其加入就绪队列

2. 切换到调度器上下文
3. 调度器选择下一个就绪进程
4. 切换到该进程上下文，继续运行

中断寄存器的保存和恢复以汇编代码形式加入任务上下文的切换代码中。即在任务切换时，`uie`, `uip`, `uepc`, `utvec`, `utval`, `ucause` 六个寄存器也被存放到栈上，等待下一次被调度时用于恢复状态。

当进程被暂停时，对于其申请的每一个外设，在**当前**硬件线程的 U 态 PLIC 上下文中将其禁用；如果用户进程启用了该外设中断，则在**当前**硬件线程的 S 态 PLIC 上下文中启用，否则禁用。当进程恢复运行时，将外设在所有硬件线程的 S 态上下文中禁用，如果该外设中断被启用，则在**当前**硬件线程的 U 态上下文中启用，否则禁用。

进程退出时，内核在**所有**硬件线程的 U 态上下文中领取并完成该进程申请的每个外设的中断，将其在 U 态上下文中禁用，并在 S 态上下文中启用，从外设中断映射表中移除该外设。

当从内核态返回用户态（由于进程切换或系统调用）时，内核读取该进程的用户态中断上下文，若缓冲区不为空，则将缓冲区中的中断数量写入 `uscratch` 寄存器，同时置位 `sip.USIP`，返回用户态。

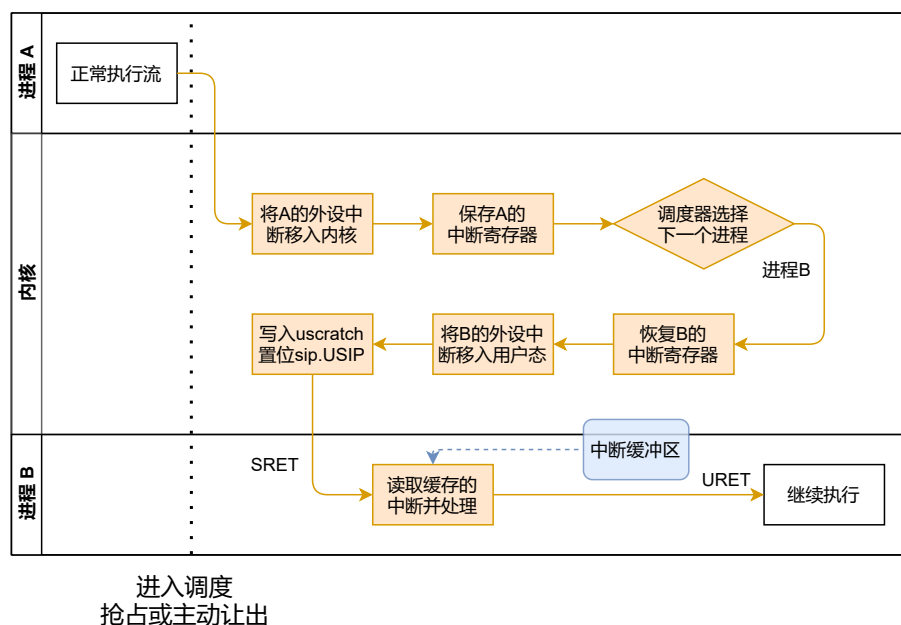


图 2.2 进程切换的处理流程

2.4.4 中断转发和注入

在 RISC-V 特权级规范^{[6]33-34}中，M 态的跨核软中断需要通过核心本地中断器 (Core-Local Interruptor, CLINT) 置位目标核上的 mip.MSIP 位，而 S 态的跨核软件中断需要通过 M 态转发，或通过其他的平台特定的中断控制器 (如 ACLINT^[18])；类似地，我们约定 U 态跨核软件中断通过 S 态转发或其他中断控制器实现。对于 S 态转发，用户进程可以通过 send_msg() 系统调用，向目标进程发送一条 `usize` 大小的信息，内核将该信息转为一条中断记录写入目标进程的中断缓冲区中。

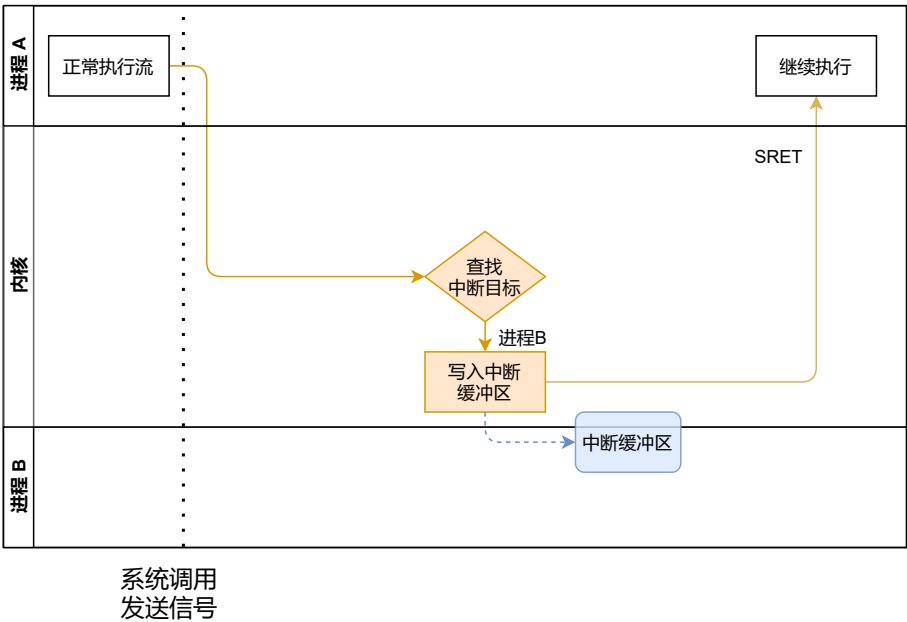


图 2.3 内核转发软件中断流程

对于时钟中断，RISC-V 特权级规范约定，硬件总是产生 M 态的时钟中断，M 态软件可以在中断处理程序中转发时钟中断到 S 态（置位 mip.STIP 并清除 mie.MTIE）。在 S 态的内核中，可以为每个硬件线程维护一个计时器队列，记录到期时刻和请求源（内核或某个进程），队列按照到期时刻由早到晚排序。

设置定时器时，将到期时间和进程 PID 写入队列中；对于内核设置的定时器，使用 0 号 PID（实际上在 rCore 中 0 号 PID 对应 initproc，但该进程不会设置定时器；或许更好的设计是使用 `usize::MAX` 指代内核）。若请求的时刻早于队列中已有的所有时刻，则（通过 SBI call）将其写入 `mtimecmp`。

内核接收到时钟中断时，将队首元素取出，判断定时器源，并更新 `mtimecmp`。若源为内核，则添加下一次调度中断，并暂停当前进程，进入调度器；若源为当前进程，则置位 sip.UTIP 并返回用户态（返回后将进入用户态中断处理函数）；若源

为另一进程，则构造一条记录（message 内容为当前时间），放入目标进程的中断缓冲区中并返回。

目前 RISC-V 社区中有关于扩展 S 态时钟中断的提议草案^[19]（即支持 S 态程序直接设置 stimecmp 寄存器等功能），未来我们也可以设计类似的用户态扩展，实现直通用户态的时钟中断。在这种情况下，utimecmp 寄存器可能需要作为进程上下文的一部分由内核协助维护。

通过上述方法实现虚拟定时器后，可以更好地支持用户态线程调度器、可抢占函数调用等功能。

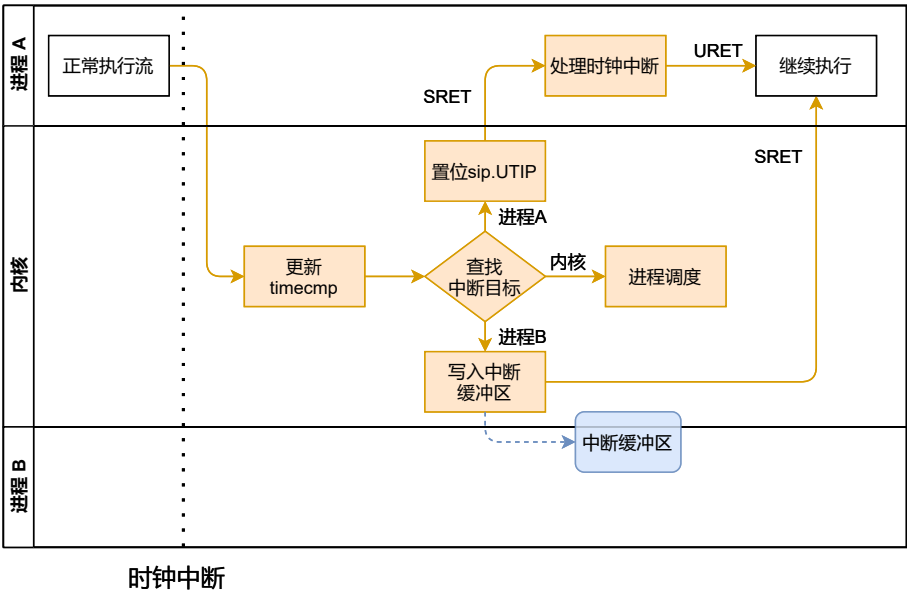


图 2.4 时钟中断处理流程

对于外部中断，由前述的进程切换流程，若某个进程正在某个硬件线程上运行，此时该进程所属的外设产生中断可以直接由 PLIC 转为 UEI 在用户态处理，而无需经过内核转发；若该进程未在运行，则 PLIC 会产生 SEI 进入内核，内核从 PLIC 领取外设编号（领取后 PLIC 会屏蔽该外设中断源信号，直至该中断被处理完成），判断应该由内核还是某个进程处理；若为后者，则向其缓冲区中写入一条中断记录并返回。用户进程的中断处理函数可以在自己的上下文中向 PLIC 提交中断完成信息，解除 PLIC 对该中断源的屏蔽。

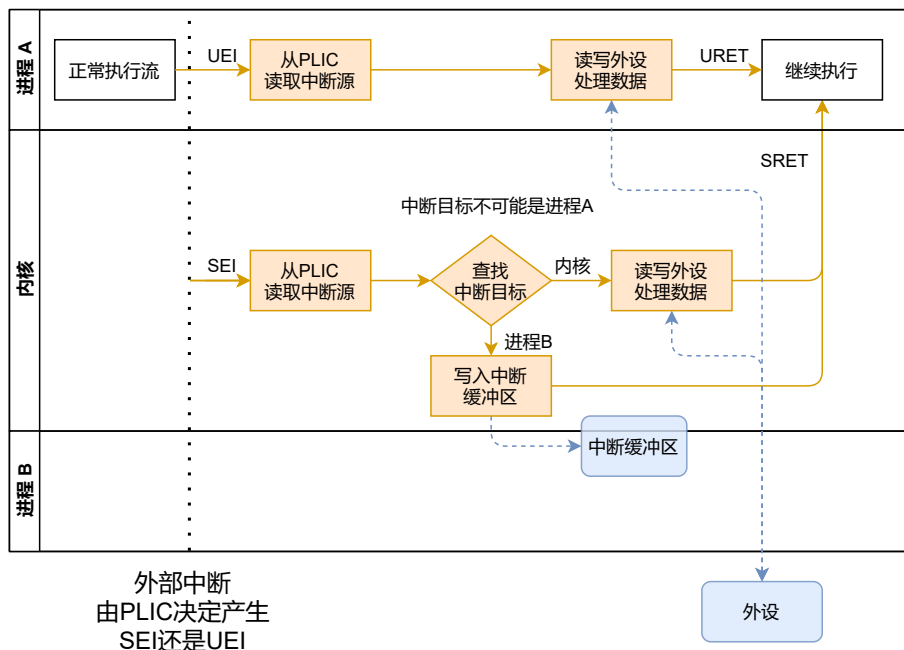


图 2.5 外部中断处理流程

2.5 应用程序接口

2.5.1 系统调用

我们在 rCore-N 中添加了五个新的系统调用，供用户进程使用用户态中断机制。

2.5.1.1 init_user_trap()

该系统调用用于通知内核为进程分配中断缓冲区内存页，并将 sstatus.UIE 置位。注意该调用不会设置 xie.UXIE，用户进程应自行设置相应位以启用中断。若无法分配内存，或进程已经调用过该函数，则本次调用会失败并返回 -1。

2.5.1.2 send_msg(pid, msg)

该系统调用将向 pid 所指定的进程发送一条消息，若发送失败则返回 -1。失败的可能原因有：目标进程未初始化中断缓冲区、缓冲区已满、目标进程全局关闭中断或相应进程不存在。这一系统调用的作用类似于 Linux 中的 signal()，但目前内核尚未对消息内容进行规范，如何解析和处理消息完全由源进程和目标进程决定。后续可能会加入对 Linux 信号编号的兼容支持，以及在内核中为一些信号提供额外的处理机制，如 SIGKILL。

2.5.1.3 set_timer(time_us)

为进程设置一个 time_us 后到期的定时器，到期后产生一个用户态时钟中断，或在相应进程的中断缓冲区中加入该信息。该调用类似于 alarm()，其效果是一次性的。

2.5.1.4 claim_ext_int(device_id)

将 device_id 对应的外设中断分配给调用进程，并将相应的 PLIC 领取/完成寄存器和外设地址映射到用户地址空间中（目前采用恒等映射）。调用成功时，返回相应外设的基址。该调用可能的失败原因较多，包括当前进程未启用中断（-1），外设地址映射失败（-2），外设编号不合法（-4），中断缓冲区未初始化（-5），PLIC 领取/完成寄存器地址映射失败（-6）。

该调用不会在用户上下文中使能相应的外设，使能需要通过下一个系统调用来实现。将这两部分分开一方面是为了允许用户程序更精细地控制每个外设单独的使能情况，也是为了避免在用户完成外设初始化之前外设触发中断。

2.5.1.5 set_ext_int_enable(device_id, enable)

在 PLIC 中启用或禁用对应外设在当前硬件线程的 U 态上下文的中断。enable > 0 表示启用，否则为禁用。若启用，会同时在所有硬件线程的 S 态的上下文中禁用该中断。调用成功时返回 0。若该外设被分配给其他进程则返回 -1，外设由内核管理时返回 -2，中断缓冲区未初始化时返回 -5。

2.5.2 用户态中断处理函数

用户进程可以读写 utvec 寄存器，令其指向自定义的中断处理函数入口。为了方便起见，我们在 rCore-N 的用户运行库中提供了一些缺省的实现，包括跳板代码、全局处理函数和三类中断各自的处理函数。

跳板代码使用汇编编写，将所有通用寄存器和一部分中断 CSR 的值保存在用户栈上，将上下文的地址写入 a0 寄存器，并跳转到全局处理函数。全局处理函数会根据 ucause 判断陷入类型，若为时钟中断，则直接调用相应的处理函数；若为外部中断，则从 PLIC 领取中断编号，传给外部中断处理函数，返回后向 PLIC 提交完成信息；若为软件中断，则从中断缓冲区中读取所有中断记录，并根据中断记录中的 cause 的值调用相应的处理函数。

全局处理函数和三类中断处理函数的签名如下：

```
#[linkage = "weak"]
```

```

#[no_mangle]
pub fn user_trap_handler(cx: &mut UserTrapContext) -> &mut
    ↪ UserTrapContext {...}

#[linkage = "weak"]
#[no_mangle]
pub fn ext_intr_handler(irq: u16, is_from_kernel: bool) {...}

#[linkage = "weak"]
#[no_mangle]
pub fn soft_intr_handler(pid: usize, msg: usize) {...}

#[linkage = "weak"]
#[no_mangle]
pub fn timer_intr_handler(time_us: usize) {...}

```

这些函数均使用弱链接标记，用户程序可以直接定义相同签名的函数，这样编译出的可执行文件中就会链接到用户的函数，而非缺省实现。注意使用 `#[no_mangle]` 标记以避免 Rust 编译器对函数重命名。

2.5.3 用户程序示例模板

```

#[no_mangle]
pub fn main() -> i32 {
    init_user_trap();
    // 从内核获取特定外设的控制权
    claim_ext_int(device_id);
    set_ext_int_enable(device_id, 1);

    // 启用对应的用户态中断
    unsafe {
        uie::set_usoft();
        uie::set_utimer();
        uie::set_uext();
    }

    ...
    // 设置时钟
    set_timer(time_us);
    // 向其他进程发送软中断
    send_msg(pid, msg);
    ...
    // 进程退出
    0
}

#[no_mangle]
pub fn ext_intr_handler(irq: u16, is_from_kernel: bool) {...}

```

```
#[no_mangle]
pub fn soft_intr_handler(pid: usize, msg: usize) {...}

#[no_mangle]
pub fn timer_intr_handler(time_us: usize) {...}
```

2.6 设计比较

本项目在中断规范方面的设计思路主要来自 N 扩展草案与 PLIC 规范，这种设计大体上是 将 RISC-V 特权级规范中关于 M 和 S 态的内容平移到了 U 态，例如 S 态有对应的 sstatus, stvec, sip, sie 等寄存器和 sret 指令，在 PLIC 中有对应的上下文。而 Intel 的 UINTR 机制设计思路则有所不同，其并未采取与 x86 中普通中断类似的中断描述符表 (Interrupt Descriptor Table, IDT)^{[4]6-1} 等方式处理，而是更类似 VT-d 扩展中用于 IO 虚拟化的中断发布 (Interrupt Posting) 机制^{[20]5-12}，后者用于将中断导入虚拟机中进行处理。例如中断发布依赖于一个名为发布中断描述符 (Posted Interrupt Descriptor, PID)^{[20]9-36} 的结构，定义如表 2.9 所示。与 UPID 的定义（表 1.2）对比，不难看出二者在结构和功能上的相似性。这一相似设计背后的逻辑在于，多个用户程序与操作系统的关系，类似于多个虚拟机与宿主机的关系。

表 2.9 PID 定义

位域	名称	描述
255:0	发布的中断请求 (PIR)	该段为一个位向量，如果某位为 1 表示对应的虚拟处理器有待处理的中断请求
256	待处理通知 (ON)	若该位被置位表示 PIR 中有一个或多个待处理的 UINTR 通知
257	抑制通知 (SN)	该位被置位时，发布到该描述符的中断不产生 UINTR 通知
271:258	预留	软件需将这些位置 0
279:272	通知向量 (NV)	用于指定通知事件使用的物理中断向量
287:280	预留	软件需将这些位置 0
319:288	通知目标 (NDST)	目标的物理 APIC ID, 由 SENDUIPI 使用。在 xAPIC 模式下，第 47:40 位为 8 位 APIC ID；在 x2APIC 模式，该域为 32 位 APIC ID
511:320	预留	软件需将这些位置 0

另一方面，Intel 的 UINTR 机制中无论是接收中断所需的 UPID 还是发送中断使用的 UITT 结构，均放于物理内存中，这样的设计在一定程度上有助于提高灵活性和扩展性，但可能提升硬件设计的复杂性和相关中断处理过程的延迟。不过考虑到 Intel 对其处理器设计有完全的话语权，这一代价应该是可以接收的。与之相对，RISC-V 指令集架构及其周边规范更注重精简设计，因此本文也延续了这一风格，没有选择将用户态中断规范相关的状态放入内存中，而是尽可能约束在寄存器和中断控制器内部。

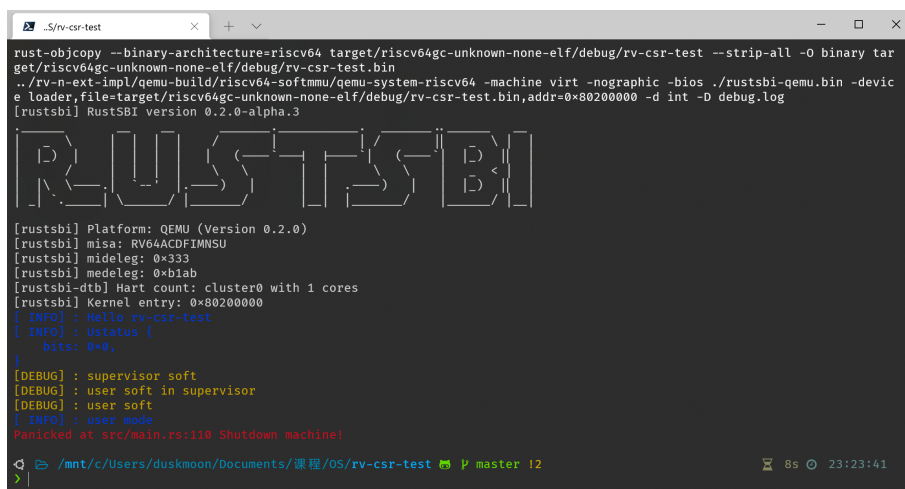
第 3 章 系统实现

3.1 模拟器与 FPGA 原型平台

为了快速测试我们的设计是否合理，我们首先基于 QEMU 模拟器^[21]实现了用户态中断机制。QEMU 是一个通用、开源的系统模拟器，能够模拟处理器、内存、外设的行为。实现的内容包括：

- 添加寄存器 `ustatus uip uie sideleg sedeleg uepc utvec ucause ut-val uscratch`
- 添加用户态中断的触发部分：符合条件时使上述处理器进入中断状态
- 实现 `uret` 指令
- 修改 PLIC 以支持用户态外部中断
- 添加多个 16550 串口用于测试

使用修改后的 QEMU 进行的用户态中断的测试结果如图 3.1 所示，图中的 `user soft` 为用户态的中断处理程序接收到 `user soft interrupt` 后输出的信息，`user mode` 为用户态程序输出的信息。



```
rust-objcopy --binary-architecture=riscv64 target/riscv64gc-unknown-none-elf/debug/rv-csr-test --strip-all -O binary target/riscv64gc-unknown-none-elf/debug/rv-csr-test.bin
../rv-n-ext-impl/qemu-build/riscv64-softmmu/qemu-system-riscv64 -machine virt -nographic -bios ./rustsbi-qemu.bin -device loader,file=target/riscv64gc-unknown-none-elf/debug/rv-csr-test.bin,addr=0x80200000 -d int -D debug.log
[rustsbi] RustSBI version 0.2.0-alpha.3

[ rustsbi ] Platform: QEMU (Version 0.2.0)
[ rustsbi ] isa: RV64ACDFIMNSU
[ rustsbi ] mideleg: 0x333
[ rustsbi ] mideleg: 0xb1ab
[ rustsbi-dtb ] Hart count: cluster0 with 1 cores
[ rustsbi ] Kernel entry: 0x80200000
[ INFO ] : Hello rv-csr-test
[ INFO ] : Ustatus {
  Bits: 0x0,
}
[DEBUG] : supervisor soft
[DEBUG] : user soft in supervisor
[DEBUG] : user soft
[ INFO ] : user mode
Panic at src/main.rs:110 Shutdown machine!
```

图 3.1 QEMU 中的用户态中断测试

由于采用软件模拟的方式，QEMU 中运行的程序并不能很好反映对应硬件机制的性能，为此我们又搭建了 FPGA 原型平台。该平台基于标签化 RISC-V 项目^[22]，使用的 CPU 核心为 Rocket Core^[23]，我们在此基础上添加了 N 扩展所需的寄存器、控制逻辑、指令和 PLIC 上下文，但并未使用和修改标签相关的部分；这些修改应当也可以平滑地迁移到其他使用 Rocket Core 的项目中。

FPGA 硬件使用赛灵思的 ZCU102 开发板^[24]，搭载 ZynqMP XCZU9EG 处理

芯片。该芯片分为处理系统（PS）和可编程逻辑（PL）两部分，前者具有四个 Arm A53 核心，运行 Linux 系统，我们使用这一部分烧写比特流、将 RISC-V 部分的操作系统和启动器二进制文件加载到 DRAM 中、复位启动 RISC-V 核心。在后者上我们实例化了四个 Rocket Core，支持 RV64IMACN 指令集、MMU、CLINT 和 PLIC，时钟频率 100MHz，具有 2MB 共享 L2 缓存和 2GB DRAM。此外还加入了若干个 16550 串口用于与 RISC-V 部分交互以及用户态驱动的演示。FPGA 硬件平台的架构如图 3.2 所示。

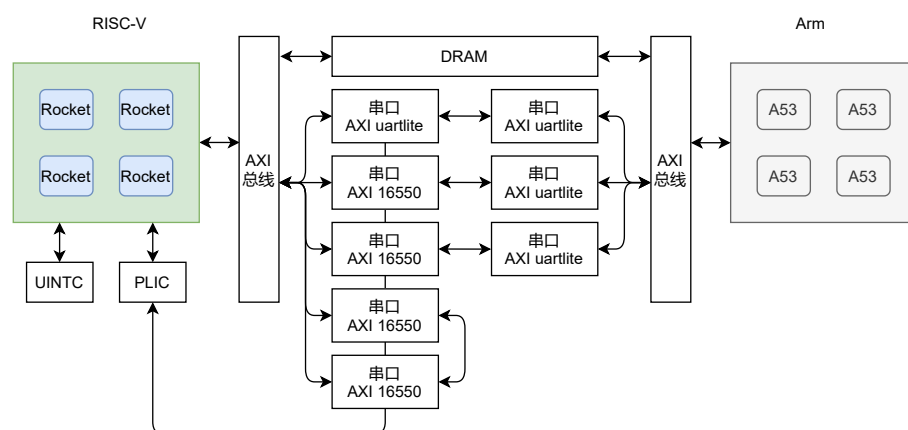


图 3.2 FPGA 硬件架构图

3.2 启动器

我们基于 `rustsbi` 库^[25] 实现 QEMU 模拟器和 FPGA 上 OS 的启动与 sbi 层服务。在 QEMU 平台上，我们在 `rustsbi-qemu` 项目^[26] 的基础上关闭了物理地址保护 (Physical Memory Protection, PMP)^[6]⁵⁶ 功能^①，以便内核和用户程序直接访问外设的地址空间。我们实现了对多核启动的支持^②，通过在启动器使用 `wfi` 指令，将核 0 以外的核卡在一个循环中，直到一个 M 态软件中断让其余核跳出循环，进入 sbi 和 os 的初始化过程。此外，还需要在启动器中将用户态的中断委托到 S 态（向 U 态的委托由内核完成）。

在 FPGA 平台上，除去地址空间布局不同需要在启动器中调整外，还需要额外处理非对齐访问异常。非对齐访问是指机器指令访问的内存地址没有对齐到指

① M 态软件可以配置 `pmpcfg` 和 `pmpaddr` 寄存器来防止 S 和 U 态程序访问或修改特定的物理地址区间。原本 `rustsbi-qemu` 只允许程序访问 sbi 部分和 OS 所在的地址空间，而并未放行 PLIC、串口等设备所在的物理地址段。

② 最新版的 `rustsbi-qemu` 已经支持了 SBI 标准的硬件线程状态管理 (Hart State Management, HSM) 扩展，可用于多核启动，但我们开发本项目时该扩展尚未完全实现。

令对应的字长边界，如 LW (Load Word，从内存中加载 4 字节的数据到寄存器中) 指令访问了一个不能被 4 整除的地址（如 0x80200003）。RISC-V 规范允许硬件在遇到非对齐访问时抛出异常，此时如果要继续执行，就需要用一系列字长更短的访存指令模拟。考虑到内核中也可能存在非对齐访存，为简化内核实现，我们将非对齐访问的处理移到了启动器中。

3.3 操作系统内核

我们基于 rCore-Tutorial v3^[27] 实现了用户态中断的支持，将新系统称为 rCore-N。rCore-Tutorial 仅支持单核，而我们的 FPGA 原型平台有 4 个处理器核，因此首先需要为其添加多核支持。

3.3.1 多核启动流程

如前文所述，我们在启动器中先启动核 0，在核 0 完成 OS 的启动和部分初始化后，通过某种方式“唤醒”其余的核。对于 0 号核，在进入 S 态后，OS 需要初始化内存、PLIC 和共用的串口外设。这些初始化工作完成后，0 号核通过调用 sbi 提供的接口，向另外三个核发送跨核中断将其他核唤醒，其他核被唤醒后根据自己的硬件线程编号进入每个核独占的初始化部分，设置自己的寄存器并配置对应自己的 PLIC 上下文。

简略的内核初始化函数如下：

```
pub fn rust_main(hart_id: usize) -> ! {
    if hart_id == 0 {
        // 全局初始化 核 0 初始化

        // 唤醒其他核
        for i in 1..CPU_NUM {
            let mask: usize = 1 << i;
            send_ipi(&mask as *const _ as usize);
        }
    } else {
        // 其他核初始化
    }
    // 开始运行
}
```

3.3.2 进程调度

出于实现和维护的简易性与简单使用场景的考虑，我们采用了单队列调度的方式。所有核共用一个“就绪进程池”，其中只有一个进程调度队列；每个核持有

一个自己正运行的进程的所有权。

```
pub struct TaskPool {  
    pub scheduler: TaskManager, // 共享的调度队列  
    ...  
}  
  
struct ProcessorInner {  
    current: Option<Arc<TaskControlBlock>>, // 该核持有的进程  
    ...  
}
```

内核初始化完成后进入调度循环，不断尝试从共享的调度队列中取出一个进程开始运行，如果没有取出则进入下一次循环。这种调度可能导致进程频繁地在核之间迁移，缓存亲和性较差，影响程序性能。为了缓解这一影响，我们在测试中额外启动了一些进程来填充空闲的处理器核。

第 4 章 性能测试与分析

4.1 测试环境

测试所用的硬件平台即为系统实现章节介绍的 FPGA 开发板。主要配置如下：

- 硬件平台：4x Rocket Core @ 100MHz, 2MB L2 Cache, 128MB DRAM
- 外设：2x AXI UART 16550 @ 6.25M baudrate

串口配置为 8 比特字长，无校验位，1 停止位，理论吞吐率为 625KB/s。吞吐率测试分为裸机环境和 rCore-N 环，而延迟测试仅在 rCore-N 环境下进行。

裸机环境经启动器初始化后进入 S 态，不开启虚拟内存，在两个核心上运行固定的程序，分别控制串口对向发送数据；S 态测试完成后构造一个简单的上下文进入 U 态，进行同样的测试过程。时钟中断配置用于控制测试结束，在测试过程中不会产生

rCore-N 环境中运行 rCore-N 操作系统，测试程序通过 read/write 系统调用访问 S 态的串口驱动，或直接获取串口对应的地址空间在用户态进行操作。此时启用虚拟内存，并配置时钟中断周期为 10 毫秒，用于执行抢占式任务调度。

4.2 驱动吞吐率测试

测试结果如表 4.1 所示，“有哈希”指每次发送或接收时均进行一次 blake3 Hasher::update() 计算，用来模拟计算和 IO 混合负载。数据单位为 KB/s。可见用户态轮询模式驱动性能最高，在裸机场景下可以接近理论上限；用户态中断模式性能次之，内核态的中断模式驱动性能最低。在混合负载的情况下，轮询模式性能下降最大，而中断模式的驱动受到的影响较小。

表 4.1 驱动吞吐率

测试配置	内核态中断模式	用户态轮询模式	用户态中断模式
裸机，无哈希	396	542	438
裸机，有哈希	123	189	136
rCore-N，无哈希	78	410	260
rCore-N，有哈希	55	152	123

4.3 驱动延时比较

4.3.1 延时测量方法

我们在程序的运行路径上插入了若干特殊的“桩”，在打桩处读取处理器的周期数并记录，在程序运行结束后对这些记录进行分析。rCore 内核所用的内存空间远小于系统的物理内存大小，且 FPGA 平台上 PS 侧的 Arm 核与 PL 侧的 RISC-V 核可以访问同一片物理内存，因此我们将这些记录写入 rCore 所用的地址段之外的特定地址上，在 PS 侧运行的 Linux 系统就可直接读出这些记录。注意由于 PL 侧的 RISC-V 核带有缓存，在读出记录前需要先将缓存内容冲刷到 DRAM 中。

4.3.2 内核驱动延时

应用程序通过 read 和 write 系统调用访问内核态驱动。read 系统调用的延时分布如图 4.1 所示，耗时集中在 15000 个周期附近；write 的延时分布如图 4.2 所示，耗时集中在 28000 个周期附近。作为对照，get_time 的调用逻辑较为简单，可以认为是一次系统调用的背景消耗，如图 4.3 所示，可见该调用延时集中在 14000 个周期附近。

内核态驱动在处理中断的耗时如图 4.4 所示，一个峰出现在 2500 周期附近且较集中，另一个峰出现在 10000 周期左右，散布较宽。

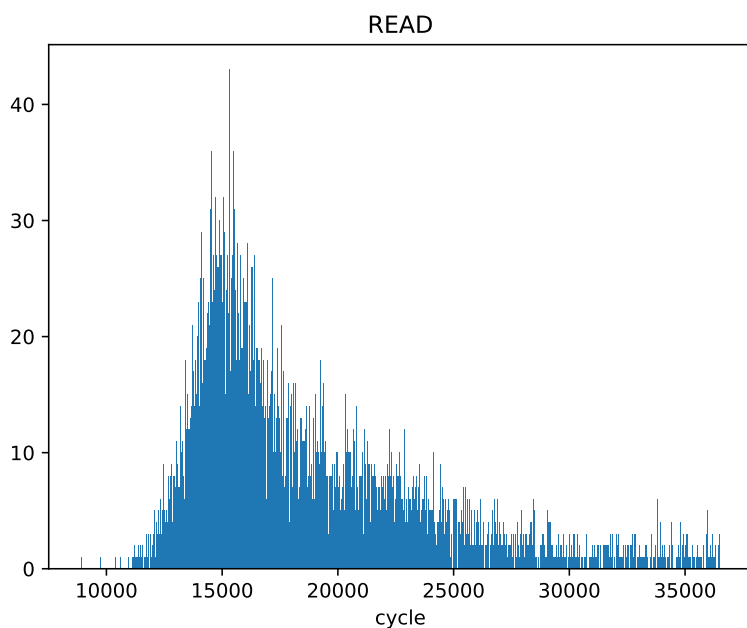


图 4.1 read 系统调用耗时分布

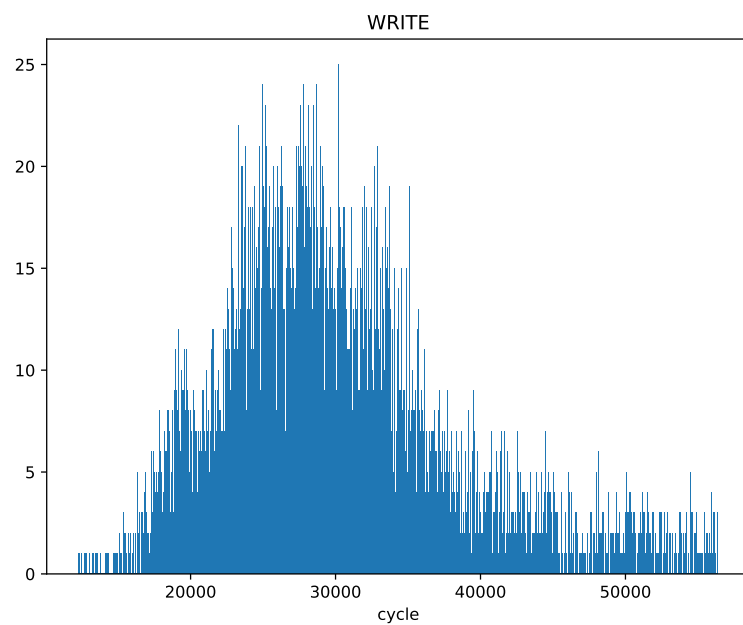


图 4.2 write 系统调用耗时分布

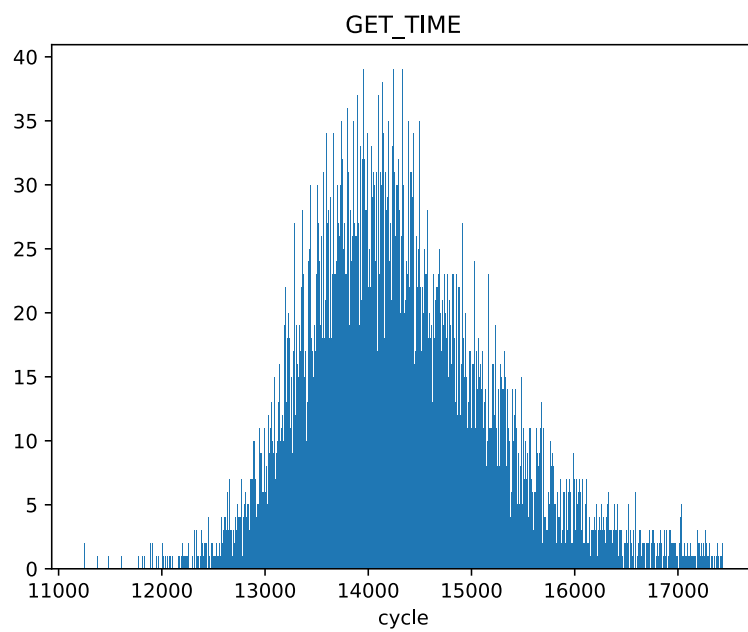


图 4.3 gettimeofday 系统调用耗时分布

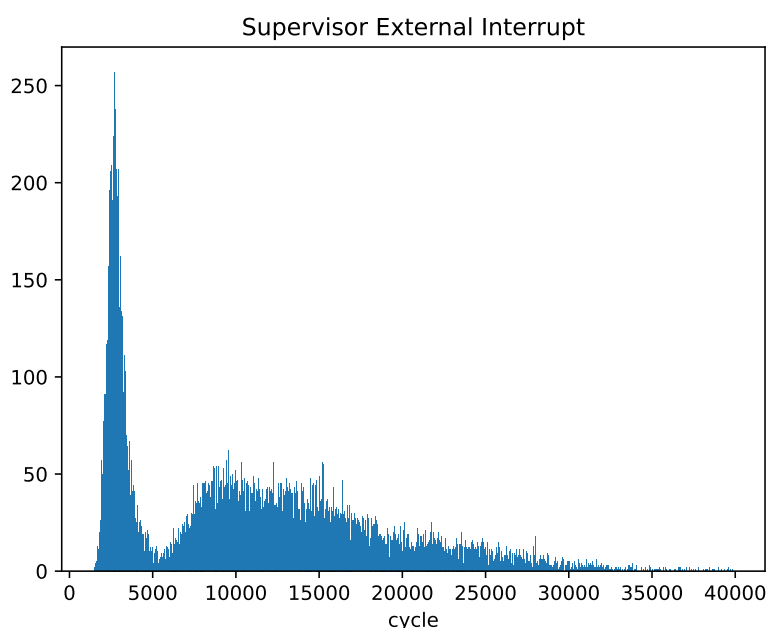


图 4.4 内核态驱动处理中断耗时分布

4.3.3 用户态中断驱动延时

调用用户态中断模式串口驱动，读或写同样数量的字符，延时分布如图 4.5 和 4.6 所示，其中读取耗时集中在 1800 和 2900 周期附近，写入延时集中在 1700 周期附近。用户态驱动处理中断的延时分布如图 4.7 所示，一个峰出现在 1800 周期附近，另一个在 4600 附近。无论是读写还是中断处理，用户态驱动所需周期数均大幅低于内核态中断驱动，且分布更集中，波动较小。这是由于用户态驱动无需切换特权级，且具有更好的代码和访存局域性。

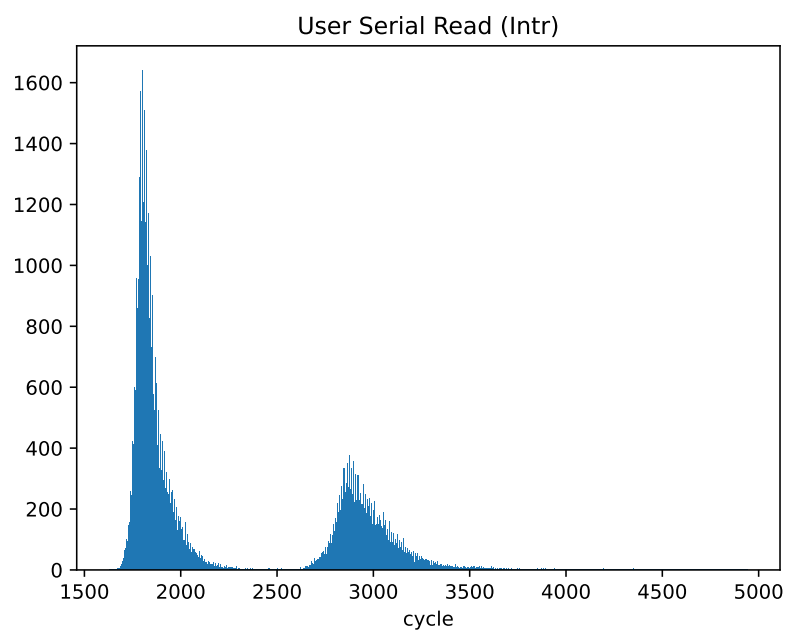


图 4.5 用户态串口驱动读取耗时分布

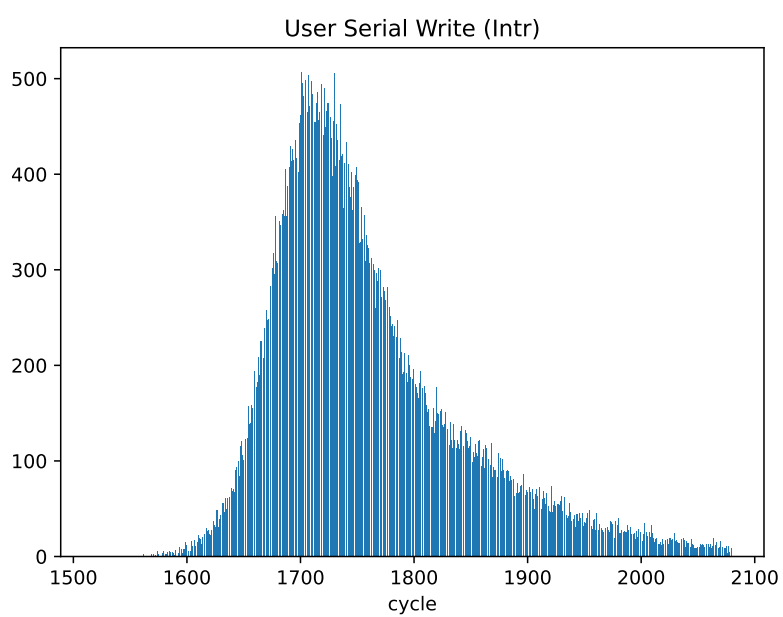


图 4.6 用户态串口驱动写入耗时分布

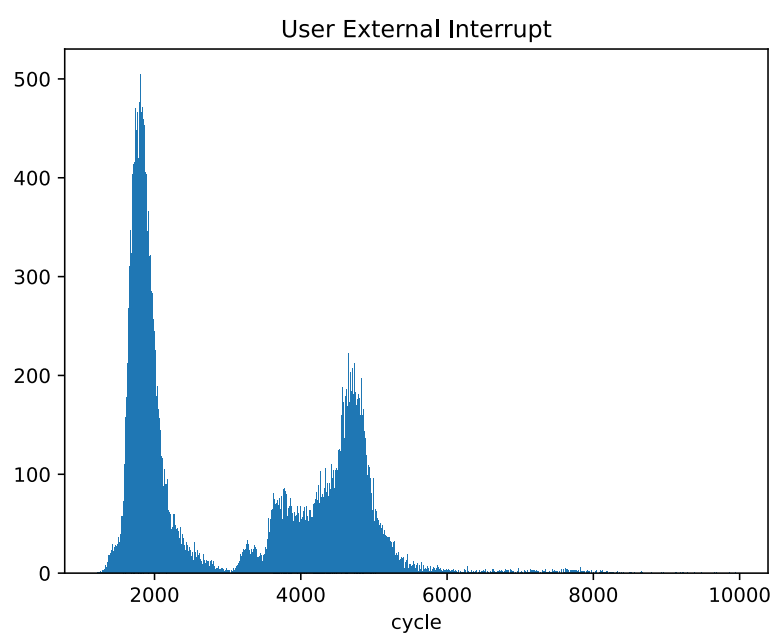


图 4.7 用户态驱动中断处理耗时分布

第 5 章 总结与展望

5.1 本文内容总结

在本文中，我们介绍了现有主流处理器的特权级和中断机制设计，分析了特权级切换对 IO 性能的影响，基于软硬件协同优化的思想，提出了一个用户态中断机制的设计和实现方案，以串口为例进行实验，证明了基于用户态中断的驱动性能提升。我们的工作主要贡献包括以下几点：

1. 完善了 RISC-V 用户态中断扩展的寄存器和指令行为规范，并提出了支持用户态外部中断的 PLIC 设计方案。我们在模拟器和 FPGA 上开发了原型平台，验证了规范和方案设计的可行性和合理性。
2. 提出了在系统软件中支持用户态中断机制的方法，基于 rCore 操作系统实现了多核多进程条件下允许用户程序复用用户态中断的管理机制。
3. 实现了基于用户态中断的外设驱动，通过实验证明其在吞吐率和延时方面的性能表现优于内核态驱动。

5.2 未来工作展望

受毕设时间和实验条件所限，本文还存在一些不完善的地方，这些部分可能在未来的工作中更进一步完善。

首先，rCore 操作系统架构和功能还较为简单，基于此实现的系统可能不太适用于实际应用场景，未来可进一步将用户态中断的支持扩展到更常用的 Linux, Zircon 等操作系统内核中。

此外，本工作仅实现了用户态外部中断的中断控制器方案，以及对用户态 IO 驱动的支持，对于用户态软件中断和时钟中断的中断控制器和应用场景探究较为不足。目前构想这两种中断可能用于优化跨进程通信、用户态线程和协程调度等情景。

最后，有一些其他实现了类似的用户态中断机制的处理器将于近期发布，如嘉楠科技的勘智 K510^[28] (使用 AX25MP 核心)、英特尔的 Sapphire Rapids 系列处理器等，但因时间等因素我们未能测试它们的实现方案并进行对比。

插图索引

图 1.1	UINTR 的处理流程	6
图 1.2	UINTR 通知的识别与处理	8
图 1.3	SENDUIPI 流程	10
图 2.1	用户态中断的处理流程	19
图 2.2	进程切换的处理流程	23
图 2.3	内核转发软件中断流程	24
图 2.4	时钟中断处理流程	25
图 2.5	外部中断处理流程	26
图 3.1	QEMU 中的用户态中断测试	31
图 3.2	FPGA 硬件架构图	32
图 4.1	read 系统调用耗时分布	36
图 4.2	write 系统调用耗时分布	37
图 4.3	gettime 系统调用耗时分布	37
图 4.4	内核态驱动处理中断耗时分布	38
图 4.5	用户态串口驱动读取耗时分布	39
图 4.6	用户态串口驱动写入耗时分布	39
图 4.7	用户态驱动中断处理耗时分布	40

表格索引

表 1.1	UINTR 相关状态定义.....	5
表 1.2	UPID 定义.....	7
表 1.3	UITTE 定义.....	9
表 2.1	ustatus 寄存器定义.....	12
表 2.2	utvec 寄存器定义.....	13
表 2.3	utvec MODE 字段定义	13
表 2.4	uip 与 uie 寄存器定义	14
表 2.5	ucause 寄存器定义.....	15
表 2.6	ucause 在陷入中断之后的值.....	16
表 2.7	URET 指令格式.....	17
表 2.8	uip 与 uie 寄存器定义	20
表 2.9	PID 定义.....	29
表 4.1	驱动吞吐率	35

参考文献

- [1] Data plane development kit[EB/OL]. LF Projects, LLC, 2022. <https://www.dpdk.org>.
- [2] Storage performance development kit[EB/OL]. Intel Corporation, 2022. <https://spdk.io>.
- [3] Kaffes K, Chong T, Humphries J T, et al. Shinjuku: Preemptive scheduling for μ second-scale tail latency[C/OL]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). Boston, MA: USENIX Association, 2019: 345-360. <https://www.usenix.org/conference/nsdi19/presentation/kaffes>.
- [4] Intel® 64 and ia-32 architectures software developer's manual - volume 3a: System programming guide, part 1[M/OL]. Intel Corporation, 2016: 5-6. <https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>.
- [5] Aarch64 exception model[EB/OL]. Arm Limited, 2022. <https://developer.arm.com/documentation/102412/0102/Privilege-and-Exception-levels>.
- [6] Waterman A, Asanović K, Hauser J. The risc-v instruction set manual, volume ii: Privileged architecture, document version 20211203[M/OL]. RISC-V International, 2021. <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>.
- [7] Intel® 64 and ia-32 architectures software developer's manual - volume 3c: System programming guide, part 3[M/OL]. Intel Corporation, 2016: 23-1. <https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html>.
- [8] Mi Z, Li D, Yang Z, et al. Skybridge: Fast and secure inter-process communication for micro-kernels[C/OL]//EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019. New York, NY, USA: Association for Computing Machinery, 2019. <https://doi.org/10.1145/3302424.3303946>.
- [9] Waterman A, Asanović K, Hauser J. The risc-v instruction set manual, volume ii: Privileged architecture, document version v1.12-draft[M/OL]. RISC-V International, 2019: 113-116. <https://github.com/riscv/riscv-isa-manual/releases/download/draft-20210612-98f3349/riscv-privileged.pdf>.
- [10] Waterman A, Heiser G, Mi Z, et al. Proposed deprecation of n extension[EB/OL]. 2021. <https://lists.riscv.org/g/tech-privileged/topic/83320504>.
- [11] 昉·天枢处理器内核[EB/OL]. 上海赛昉科技有限公司, 2020[2022-05-26]. <https://starfive.tech.com/site/riscv-core-ip>.

- [12] Andescore™ ax25mp multicore[EB/OL]. 晶心科技股份有限公司, 2019[2022-05-26]. <http://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax25mp>.
- [13] Gala N, Jagan L, Sarma D N, et al. Shakti c-class core generator[EB/OL]. IIT Madras, 2020. <https://gitlab.com/shaktiproject/cores/c-class>.
- [14] Intel® architecture instruction set extensions and future features programming reference [M/OL]. Intel Corporation, 2021: 11-1. <https://www.intel.com/content/www/us/en/development/download/intel-architecture-instruction-set-extensions-programming-reference.html>.
- [15] Mehta S. x86 user interrupts support[EB/OL]. 2021. <https://lore.kernel.org/lkml/20210913200132.3396598-1-sohil.mehta@intel.com>.
- [16] Mehta S. uintr-linux-kernel[EB/OL]. 2021. <https://github.com/intel/uintr-linux-kernel/tree/rfc-v1>.
- [17] Barbier D, Dabbelt P, Chang A. Risc-v platform-level interrupt controller specification[M/OL]. 2022. <https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc>.
- [18] Waterman A, Favor G, Hauser J, et al. Risc-v advanced core local interruptor specification [M/OL]. 2022. <https://github.com/riscv/riscv-aclint/blob/main/riscv-aclint.adoc>.
- [19] Scheid J, Scheel J. Risc-v "stimecmp / vstimecmp" extension: Sstc[M/OL]. 2021. <https://github.com/riscv/riscv-time-compare>.
- [20] Intel® virtualization technology for directed i/o architecture specification[M/OL]. Intel Corporation, 2022. <https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf>.
- [21] Bellard F, the QEMU team. Qemu[EB/OL]. 2022[2022-05-26]. <https://www.qemu.org>.
- [22] Yu Z, Huang B, Ma J, et al. Labeled risc-v: A new perspective on software-defined architecture [C/OL]//CARRV 2017: Proceedings of Computer Architecture Research with RISC-V. Boston, MA, USA, 2017. https://carrv.github.io/2017/papers/yu-labeled_riscv-carrv2017.pdf. DOI: 10.475/123_4.
- [23] Asanović K, Avizienis R, Bachrach J, et al. The rocket chip generator: UCB/EECS-2016-17 [R/OL]. EECS Department, University of California, Berkeley, 2016. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [24] Zynq ultrascale+ mp soc zcu102 evaluation kit[EB/OL]. Xilinx, Inc, 2022[2022-05-26]. <https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html>.
- [25] Luo J, Wu Y, He K, et al. Rustsbi: A risc-v supervisor binary interface (sbi) implementation in rust[EB/OL]. 2022[2022-05-17]. <https://github.com/rustsbi/rustsbi>.
- [26] Luo J, Wu Y, He K, et al. Qemu support from rustsbi[EB/OL]. 2022[2022-05-10]. <https://github.com/rustsbi/rustsbi-qemu>.

- [27] Wu Y, Chen Y, Zhang Y, et al. rcore-tutorial-v3[EB/OL]. 2022[2022-05-20]. <https://github.com/rcore-os/rCore-Tutorial-v3>.
- [28] 勘智 K510[EB/OL]. 嘉楠科技, 2022[2022-05-26]. <https://canaan-creative.com/product/%E5%8B%98%E6%99%BAk510>.

致 谢

首先我要感谢马洪兵老师和陈渝老师两位导师。在毕设选题之初，陈渝老师对研究方向和目标提供了很多有价值的参考，中期时两位老师高度负责地对我的进度进行了细致的监督和检查，为中期报告提出了宝贵的修改意见，结题阶段对我的论文写作和答辩报告也提供了很多建议。

其次，我要感谢向勇老师和贺鲲鹏同学。在前期确定系统规范的阶段，他们和我进行了非常丰富的交流讨论，这些交流为整个系统后续的开发和实现奠定了基础，令我受益匪浅；在系统移植和模拟器实现等阶段，他们也提供了极大的帮助。

接着我要感谢张译仁和吴一凡学长、华中科技大学的蒋周奇和车春池同学、中科院计算所的张传奇博士，他们为操作系统调试、启动器设计、硬件平台开发等方面提供了大量的支持，在与他们的交流中我获得了许多启发和灵感。

此外，我还想感谢我的父母、朋友、室友、同学、老师们，以及其他所有曾帮助和指点过我的人，感谢他们对我的支持和鼓励。

最后我想感谢我的笔记本电脑，它不眠不休坚持完成了多次持续数个小时的高强度 **FPGA** 逻辑综合和实现任务，轻载不响风扇，重载不烧供电，尽管多次蓝屏，但都避开了项目的关键节点，为整个项目的顺利完成提供了重要保障。

附录 A 外文资料的书面翻译

操作系统大冒险：用 Rust 语言编写一个 RISC-V 操作系统 (第 3-4 章)

目录

A.1 页粒度的内存分配	49
A.1.1 概述	49
A.1.2 以页为单位的分配	49
A.1.3 描述符的分配	50
A.1.4 页面分配	51
A.1.5 页面释放	52
A.1.6 分配清零的页面	53
A.1.7 测试	53
A.1.8 未来	55
A.2 内存管理单元	56
A.2.1 概述	56
A.2.2 分页	56
A.2.3 虚拟地址	56
A.2.4 页表项	57
A.2.5 物理地址	57
A.2.6 SATP 寄存器	57
A.2.7 页面错误	58
A.2.8 虚拟地址翻译	58
A.2.9 每一级页表都可以成为叶项	59
A.2.10 对 MMU 的编程	59
A.2.11 释放 MMU 表	62
A.2.12 遍历页表	63
A.2.13 A 和 D 位	65
A.2.14 内核的地址映射	65
A.2.15 开启 MMU	68

A.3 处理中断和陷入	69
A.3.1 概述	69
A.3.2 RISC-V 中断系统.....	69
A.3.3 为什么陷入?	70
A.3.4 简单启动	72
A.3.5 陷入	72
A.3.6 陷入帧 Rust 结构体	75
A.3.7 查看汇编中的陷入向量	76
A.3.8 GNU 汇编宏	77
A.3.9 这有什么作用?	78
参考文献.....	79

A.1 页粒度的内存分配

A.1.1 概述

当我们启动时，`.elf` 文件内的所有内容都会自动加载到内存中。在代码中可见，我们为整个系统分配了 128 兆字节空间。因此，我们需要处理除去所有声明的全局变量和代码之外的空间。

在管理内存时，我们本质上需要处理三个部分：1) 页粒度的分配，2) 字节粒度的分配，以及 3) 内存管理单元的编程。

A.1.2 以页为单位的分配

以页为单位的分配意味着每一次分配所提供的内存大小是一个页。在大多数架构中，最小的页大小是 4096 字节。RISC-V 也不例外。我们将使用 SV39 MMU 分页系统，这些将在第 3 部分中深入讨论。

我们有多种方法可以一次分配页面。由于我们知道每一次分配的大小，分配方案就很直白了。许多操作系统使用一个链表进行分配，其中指向链表头部的指针是第一个未分配的页面，然后每个未分配的页面用 8 个字节来存储下一个指针，指向下一个页面的内存地址。这些字节是存储在页面本身之中的，所以不需要额外的内存的开销。

这个系统的缺点是，我们必须跟踪每一个被分配的页面，因为一个指针就是一个页面。

```
struct FreePages {
```

```
    struct FreePages *next;
};
```

A.1.3 描述符的分配

我们的操作系统将使用描述符。我们可以分配连续的页面，而只存储链表头的指针。然而要做到这一点，我们需要为每个 4096 字节大小的页额外分配一个字节。这一个字节将包含两个标志：1) 这个页面是否被占用？2) 这是不是连续分配的最后一个页面？

```
// These are the page flags, we represent this
// as a u8, since the Page stores this flag.
#[repr(u8)]
pub enum PageBits {
    Empty = 0,
    Taken = 1 << 0,
    Last = 1 << 1,
}

pub struct Page {
    flags: u8,
}
```

在这里，我们需要为每个页面分配一个 Page 结构体。幸运的是，通过升级过的链接器脚本，我们能够获得堆的地址和大小。

```
/*
Finally, our heap starts right after the kernel stack. This
    ↪ heap will be used mainly
to dole out memory for user-space applications. However, in
    ↪ some circumstances, it will
be used for kernel memory as well.

We don't align here because we let the kernel determine how it
    ↪ wants to do this.
*/
PROVIDE(_heap_start = _stack_end);
PROVIDE(_heap_size = _memory_end - _heap_start);
```

我们可以简单地计算出堆中的总页数：let num_pages = HEAP_SIZE /
↪ PAGE_SIZE;，其中 HEAP_SIZE 是一个全局符号，值为 _heap_size，页大小是一个常数为 4096。最终我们必须分配 num_pages 数量的 Page 描述符来存储分配信息。

A.1.4 页面分配

分配是通过首先在描述符链表中搜索一个连续、空闲的页面块来完成的。我们接收想要分配的页数量作为 `alloc` 的参数。如果我们找到一段空闲的页，我们就把所有这些 `Page` 的描述符都设置为已占用，然后将最后一个页的“Last”位置位。这可以帮助我们在释放分配的页面时，跟踪最后一个页面的位置。

```
pub fn alloc(pages: usize) -> *mut u8 {
    // We have to find a contiguous allocation of pages
    assert!(pages > 0);
    unsafe {
        // We create a Page structure for each page on the heap.
        // ↪ We
        // actually might have more since HEAP_SIZE moves and so
        // ↪ does
        // the size of our structure, but we'll only waste a few
        // ↪ bytes.
        let num_pages = HEAP_SIZE / PAGE_SIZE;
        let ptr = HEAP_START as *mut Page;
        for i in 0..num_pages - pages {
            let mut found = false;
            // Check to see if this Page is free. If so, we have
            // ↪ our
            // first candidate memory address.
            if (*ptr.add(i)).is_free() {
                // It was FREE! Yay!
                found = true;
                for j in i..i + pages {
                    // Now check to see if we have a
                    // contiguous allocation for all of the
                    // request pages. If not, we should
                    // check somewhere else.
                    if (*ptr.add(j)).is_taken() {
                        found = false;
                        break;
                    }
                }
            }
        }
        // .....
    }
}
```

我们可以简单算出哪个描述符指向哪个页面，因为第一个描述符就是第一页，第二个描述符是第二页，...，第 `n` 个描述符是第 `n` 页。这种一对一的关系简化了数学运算：`ALLOC_START + PAGE_SIZE * i`。在这个公式中，`ALLOC_START` 是第一个可用页面的地址，`PAGE_SIZE` 是常数 4096，而 `i` 是指第 `i` 个页面描述符（从 0 开始）。

A.1.5 页面释放

首先我们必须把上面的等式倒过来，计算出这个页面对应的是哪个描述符。我们可以使用这个式子： $\text{HEAP_START} + (\text{ptr} - \text{ALLOC_START}) / \text{PAGE_SIZE}$ ，我们接收一个指向可使用页面的指针 `ptr`，我们从该指针中减去可使用的堆顶部地址，以获得该 `ptr` 的绝对位置。由于每次分配都正好是 4096 字节，我们再除去页面大小来得到描述符下标。然后我们再加上 `HEAP_START` 变量，这样就得到了描述符的位置。

```
pub fn dealloc(ptr: *mut u8) {
    // Make sure we don't try to free a null pointer.
    assert!(!ptr.is_null());
    unsafe {
        let addr =
            HEAP_START + (ptr as usize - ALLOC_START) / PAGE_SIZE;
        // Make sure that the address makes sense. The address
        // ↪ we
        // calculate here is the page structure, not the HEAP
        // ↪ address!
        assert!(addr >= HEAP_START && addr < HEAP_START +
            // ↪ HEAP_SIZE);
        let mut p = addr as *mut Page;
        // Keep clearing pages until we hit the last page.
        while (*p).is_taken() && !(*p).is_last() {
            (*p).clear();
            p = p.add(1);
        }
        // If the following assertion fails, it is most likely
        // caused by a double-free.
        assert!(
            (*p).is_last() == true,
            "Possible double-free detected! (Not taken found \
                before last)"
        );
        // If we get here, we've taken care of all previous
        // ↪ pages and
        // we are on the last page.
        (*p).clear();
    }
}
```

在这段代码中，我们可以看到，我们一直释放空间直到最后一个被分配的页。如果我们在到最后一个分配的页面之前遇到了一个没有被占用的页面，那么我们的堆就乱套了。通常情况下，当一个指针被释放超过一次时就会发生这种情况（常称为双重释放）。在这种情况下我们可以添加一个 `assert!` 语句来帮助捕捉这些类型的问题，但我们必须使用“可能”一词，因为我们不能确定为什么在出现非占

用页之前没有遇到最后一个被分配的页。

A.1.6 分配清零的页面

这些 4096 字节的页将分配给内核和用户应用使用。注意当我们释放一个页面时，我们并不清除内存内容，相反我们只清除描述符。在上述代码中这表现为 `(*p).clear()`，这段代码只清除了描述符对应的位，而不是整个页的 4096 字节。

安全起见，我们将创建一个辅助函数，分配并清除这些页面。

```
/// Allocate and zero a page or multiple pages
/// pages: the number of pages to allocate
/// Each page is PAGE_SIZE which is calculated as 1 <<
    ↪ PAGE_ORDER
/// On RISC-V, this typically will be 4,096 bytes.
pub fn zalloc(pages: usize) -> *mut u8 {
    // Allocate and zero a page.
    // First, let's get the allocation
    let ret = alloc(pages);
    if !ret.is_null() {
        let size = (PAGE_SIZE * pages) / 8;
        let big_ptr = ret as *mut u64;
        for i in 0..size {
            // We use big_ptr so that we can force an
            // sd (store doubleword) instruction rather than
            // the sb. This means 8x fewer stores than before.
            // Typically we have to be concerned about remaining
            // bytes, but fortunately 4096 % 8 = 0, so we
            // won't have any remaining bytes.
            unsafe {
                (*big_ptr.add(i)) = 0;
            }
        }
    }
}
```

我们简单地调用 `alloc` 来完成那些繁重的工作——分配页面描述符并设置标志位。然后我们进入 `zalloc` 的后半部分，即将页面清零。幸运的是，我们可以使用 8 字节的指针来一次性存储 8 字节的 0，因为 $4096 / 8 = 512$ ，而 512 是一个整数。因此，8 字节的指针只需 512 轮循环即可完全覆盖一个 4096 字节的页面。

A.1.7 测试

我创建了一个函数，通过查看所有占用的描述符来打印出页分配表。

```
/// Print all page allocations
/// This is mainly used for debugging.
pub fn print_page_allocations() {
    unsafe {
```

```

let num_pages = HEAP_SIZE / PAGE_SIZE;
let mut beg = HEAP_START as *const Page;
let end = beg.add(num_pages);
let alloc_beg = ALLOC_START;
let alloc_end = ALLOC_START + num_pages * PAGE_SIZE;
println!();
println!(
    "PAGE ALLOCATION TABLE\nMETA: {:p} -> {:p}\n
    ↪ nPHYS: \
    0x{:x} -> 0x{:x}",
    beg, end, alloc_beg, alloc_end
);
println!("~~~~~");
let mut num = 0;
while beg < end {
    if (*beg).is_taken() {
        let start = beg as usize;
        let memaddr = ALLOC_START
            + (start - HEAP_START)
            * PAGE_SIZE;
        print!("0x{:x} => ", memaddr);
        loop {
            num += 1;
            if (*beg).is_last() {
                let end = beg as usize;
                let memaddr = ALLOC_START
                    + (end
                        - HEAP_START)
                    * PAGE_SIZE
                    + PAGE_SIZE - 1;
                print!(
                    "0x{:x}: {:>3} page(s) ",
                    memaddr,
                    (end - start + 1)
                );
                println!(".");
                break;
            }
            beg = beg.add(1);
        }
        beg = beg.add(1);
    }
    num = num.add(1);
}
println!("~~~~~");
println!(
    "Allocated: {:>6} pages ({:>10} bytes).",
    num,
    num * PAGE_SIZE
);
println!(
    "Free      : {:>6} pages ({:>10} bytes).",

```

```

        num_pages - num,
        (num_pages - num) * PAGE_SIZE
    );
    println!();
}
}

```

然后，让我们来分配一些页面！我已经分配了一些单页和一段较大的 64 页空间。你会看到如图 A.1 打印出的内容。

```

PAGE ALLOCATION TABLE
META: 0x8004e0d8 -> 0x80056089
PHYS: 0x80057000 -> 0x88008000
~~~~~
0x80057000 => 0x80096fff: 64 page(s).
0x80097000 => 0x80097fff: 1 page(s).
0x80098000 => 0x80098fff: 1 page(s).
0x80099000 => 0x80099fff: 1 page(s).
0x8009a000 => 0x8009afff: 1 page(s).
0x8009b000 => 0x8009bfff: 1 page(s).
0x8009c000 => 0x8009cfff: 1 page(s).
0x8009d000 => 0x8009dfff: 1 page(s).
0x8009e000 => 0x8009efff: 1 page(s).
~~~~~
Allocated: 72 pages ( 294912 bytes).
Free      : 32617 pages (133599232 bytes).

```

图 A.1 页分配输出结果

A.1.8 未来

对于较小的数据结构，页粒度的分配器会浪费大量的内存。例如，一个存储整数的链表结点将需要 4 字节 + 8 字节 = 12 字节的内存。4 字节用于整数，8 字节用于下一个指针。然而我们必须动态分配一整页。因此，我们有可能浪费 $4096 - 12 = 4084$ 字节。所以，我们可以做的是更精细管理每个页面的分配，这个以字节为单位的分配器将很像 C++ 中 `malloc` 的实现方式。

当我们开始分配用户空间的进程时，我们需要保护内核内存不被用户的应用程序错误地写入，因此我们将使用内存管理单元，特别地，我们将使用 Sv39 方案，它的文档在 RISC-V 特权规范^[1] 第 4.4 章中。

本章还将关注两个部分：2) 内存管理单元 (MMU) 和 3) Rust 中 `allocate`

的字节粒度内存分配器，它包含数据结构，如链表和二叉树。我们还可以使用这个 `crate` 来存储和创建 `String` 类型。

A.2 内存管理单元

A.2.1 概述

从操作系统的角度来看，我们有一个从某个起始地址到某个结束地址的大内存池。我们可以使用链接脚本（`virt.lds` 文件）提供表示起始和终止地址的符号。

一些内存将用于全局变量、堆栈空间等。但是，我们使用链接脚本来帮助我们处理这些不断移动的东西。其余空闲的内存成为我们的“堆”空间。然而，操作系统中的堆并不完全是 `malloc`, `free` 所使用的堆。

A.2.2 分页

分页是一个通过硬件（通常称为内存管理单元或 MMU）将虚拟地址翻译为物理地址的系统。MMU 通过读取名为 SATP（Supervisor Address Translation and Protection）的特权寄存器来执行地址翻译。该寄存器控制 MMU 的使能，设置地址空间标识符，并设置第一级页表所在的物理内存地址。

我们可以将这些表放在 RAM 中的任何位置，只要地址最后 12 位为 0。这是因为 SATP 中没有提供页表的最后 12 位。事实上，为了获得实际的内存地址，我们从 SATP 中取出 44 位并将其左移 12 位，同时在低 12 位填充 0。这组成了一个 56 位的地址——而不是 64 位。

本质上我们只需要关注以下三点：(1) 虚拟地址，(2) 物理地址，以及 (3) 页表项。这些都列在 RISC-V 特权规范^[1] 第 4.4 章中。

RISC-V 有不同的分页模式。HiFive Unleashed 使用 Sv39 模式，这意味着虚拟地址为 39 位。在这种情况下，虚拟地址无法使用完整的 64 位内存空间。其他的分页模式包括使用 32 位虚拟地址的 Sv32 和使用 48 位虚拟地址的 Sv48。

A.2.3 虚拟地址

Sv39 虚拟地址包含三个 9 位索引。由于 $2^9 = 512$ ，每级页表都包含 512 个页表项，每个页表项正好是 8 字节（64 位），如图 A.2 所示。Sv39 虚拟地址包含 `VPN[x]`。VPN 代表“虚拟页号”，它本质上是一个包含 512 个 8 字节条目的数组的索引。因此，MMU 不是直接将虚拟地址转换为物理地址，而是通过一系列页表。使用 Sv39 系统，我们的页表可以三个级别，每个级别是一张包含 512 个 8 字节项

的表。

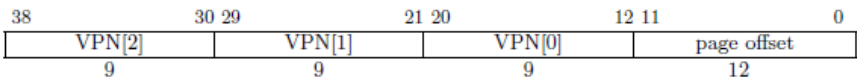


Figure 4.16: Sv39 virtual address.

图 A.2 SV39 虚拟地址

A.2.4 页表项

操作系统会写入一个页表项来控制 MMU 的工作方式。我们可以更改地址的翻译方式，也可以设置某些位来保护页面。页面入口图 A.3 所示：

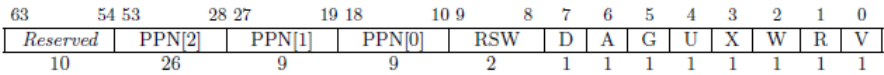


Figure 4.18: Sv39 page table entry.

图 A.3 SV39 页表项

A.2.5 物理地址

物理地址实际上是 56 位。因此，一个 39 位的虚拟地址可以转换为一个 56 位的物理地址。显然，这允许我们将相同的虚拟地址映射到不同的物理地址——就像我们在创建用户进程时映射地址的方式一样。物理地址是通过获取 PPN[2:0] 并将它们按照如图 A.4 格式拼接来形成的。可以注意到页面偏移量是直接从虚拟地址复制到物理地址的。

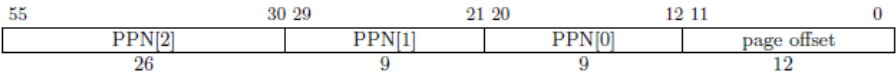


Figure 4.17: Sv39 physical address.

图 A.4 SV39 物理地址

A.2.6 SATP 寄存器

所有的地址翻译都从如下所示的 SATP 寄存器开始，具体描述在 RISC-V 特权规范^[1] 第 4.1.12 章中。SATP 寄存器的格式如图 A.5 所示。

我们可以看到 PPN（物理页号）是一个 44 位的值。但是，该值在存储之前已右移 12 位，这本质上是将实际地址除以 4096 ($2^{12} = 4096$)。因此，实际地址为 $PPN \ll 12$ ，

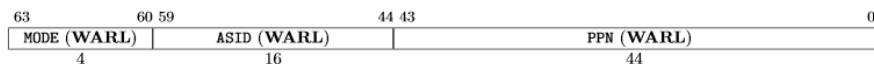


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

图 A.5 satp 寄存器

A.2.7 页面错误

MMU 通过三个不同的分页错误向 CPU 发出地址翻译错误信号：(1) 指令、(2) 加载和 (3) 存储。当 MMU 引发页面错误时，在取指阶段发生指令分页错误。这可能意味着页表项无效或没有将 X（执行）位设置为 1 或其他原因。

页面错误由 CPU 捕获，`mcause` 或 `scause` 寄存器中存有异常编号。12 表示指令页面错误、13 表示加载页面错误、15 表示存储页面错误。操作系统可以决定如何处理。在大多数情况下，正确的处理方式是杀死有问题的进程。但是，有一些高级的系统机制，例如写时复制（Copy-on-Write, CoW），是通过捕获页面错误来实现的。在这些情况下，页面错误不是一个问题，而是一个特性。

A.2.8 虚拟地址翻译

举个例子。虚拟地址 `0x7d_beef_cafe`，正好是 39 位。如果地址长于 39 位，MMU 很可能会产生一个页面错误。

二进制：0b0111_1101_1011_1110_1110_1111_1100_1010_1111_1110，各部分如表 A.1 所示。

表 A.1 虚拟地址翻译示例

VPN[2]	VPN[1]	VPN[0]	12 位偏移
1_1111_0110	1_1111_0111	0_1111_1100	1010_1111_1110
502	503	252	

上面的地址将按照以下步骤进行翻译：

1. 读取 SATP 寄存器并找到第 2 级页表的首项地址 ($PPN \ll 12$)。2. 在首项地址上加上偏移量大小，其中偏移量为 $VPN[2] = 0b1_1111_0110 = 502 * 8 \rightarrow SATP + 4016$ 。3. 读取此条目。4. 如果 $V(valid)=0$ ，产生页面错误 5. 如果该条目的 R、W 或 X 位为 1，则该页表项为叶项，否则为枝干项。6. $PPN[2] | PPN[1] | PPN[0]$ 地址指示下一个页表在物理内存中的位置。7. 从第 2 步重复，直到找到叶项。8. 叶项表示 $PPN[2]$ 、 $PPN[1]$ 和 $PPN[0]$ 告诉 MMU 实际物理地址是什么。

A.2.9 每一级页表都可以成为叶项

像大多数内存管理单元一样,MMU 可以翻译更粗粒度的页面。例如,Intel/AMD x86_64 系统可以在 PDP 处停止,这可以提供 1GB 的粒度,而在 PD 处停止将提供 2MB 的粒度。这同样适用于 RISC-V 系统。不过, RISC-V 采用了一种巧妙的方法来区分叶项和枝干项。

在 Intel/AMD 机器上,上级页表控制下级页表。如果某个上级页表是只读的,那么之后的任何枝干项都不可写。RISC-V 认为这很愚蠢,所以他们的设计是这样的:如果 R、W、X (读、写、执行) 保护位中的任何一个被置位,则该页表项是一个页项。

如果一个页表项 (PTE) 是叶子,则 PPN[2:0] 表示物理地址。但是,如果一个 PTE 是枝干项,则 PPN[2:0] 描述可以在物理 RAM 中找到下一个页表的位置。需要注意的一点是,只有 PPN[2:leaf-level] 将用于开发物理地址。例如,如果第 2 级 (顶层) 的页表条目是叶子,那么只有 PPN[2] 对物理地址有贡献。VPN[1] 被复制到 PPN[1], VPN[0] 被复制到 PPN[0], 并且页偏移被复制, 正常。

A.2.10 对 MMU 的编程

在 Rust 中,我创建了三个对 MMU 进行编程的函数: map 、 unmap 和 virt_to_phys 。我们需要能够手动遍历页表,因为当用户空间的应用程序进行系统调用时,每个指针都是指向虚拟内存地址的指针-这与我们在内核中的地址不一样。因此,我们必须一路深入到一个物理地址,以使用户空间应用程序和内核使用的是同一个内存地址。

我使用了一些结构体来简化页表项的编程。

```
pub struct Table {  
    pub entries: [Entry; 512],  
}  
  
impl Table {  
    pub fn len() -> usize {  
        512  
    }  
}
```

第一个结构体是 Table 。它描述了一级 4096 字节的页表。这个大小来自 512 个 8 字节的页表项。一个页表项被描述为:

```
pub struct Entry {  
    pub entry: i64,  
}
```

```

impl Entry {
    pub fn is_valid(&self) -> bool {
        self.get_entry() & EntryBits::Valid.val() != 0
    }

    // 首位 (下标 #0) 是有效位 (Valid, V)
    pub fn is_invalid(&self) -> bool {
        !self.is_valid()
    }

    // 页项的 RWX 位至少有一位被置位
    pub fn is_leaf(&self) -> bool {
        self.get_entry() & 0xe != 0
    }

    pub fn is_branch(&self) -> bool {
        !self.is_leaf()
    }

    pub fn set_entry(&mut self, entry: i64) {
        self.entry = entry;
    }

    pub fn get_entry(&self) -> i64 {
        self.entry
    }
}

```

本质上我只是把 `i64` 数据类型重命名为 `Entry`，这样我就可以给它添加一些辅助函数。

`map` 函数接收一个页表根的可变引用，一个虚拟地址，一个物理地址，保护位，以及这个地址应该被映射到哪个级别。通常情况下，我们将所有的页面映射到 0 级，也就是 4KiB 级。然而，我们可以用 2 级表示比 1GiB 页，1 表示 2MiB 页，或 0 表示 4KiB 页。

```

pub fn map(root: &mut Table, vaddr: usize, paddr: usize, bits:
    ↪ i64, level: usize) {
    // 确保 RWX 至少有一位被置位，否则会导致内存泄漏并且产生一个页错误
    assert!(bits & 0xe != 0);
    // 从虚拟地址中提取 VPN。虚拟地址中，每段 VPN 都恰好是 9 位，
    // 所以我们用 0x1ff = 0b1_1111_1111 (9 位) 作为掩码
    let vpn = [
        // VPN[0] = vaddr[20:12]
        (vaddr >> 12) & 0x1ff,
        // VPN[1] = vaddr[29:21]
        (vaddr >> 21) & 0x1ff,
        // VPN[2] = vaddr[38:30]

```

```

        (vaddr >> 30) & 0x1fff,
    ];

    // 与虚拟地址类似，提取物理页号 (PPN)。但是 PPN[2] 是 26 位而不是 9
    // 位，
    // 这是不同之处。因此我们使用
    // 0x3ff_ffff = 0b11_1111_1111_1111_1111_1111 (26 位)。
    let ppn = [
        // PPN[0] = paddr[20:12]
        (paddr >> 12) & 0x1fff,
        // PPN[1] = paddr[29:21]
        (paddr >> 21) & 0x1fff,
        // PPN[2] = paddr[55:30]
        (paddr >> 30) & 0x3ff_ffff,
    ];

```

我们在这里做的第一件事是分解虚拟地址和物理地址。注意，我们并不关心页面偏移量-这是因为我们并不存储页面偏移量。相反，当 MMU 翻译一个虚拟地址时，它直接将页面偏移量复制到物理地址中，形成一个完整的 56 位物理地址。

```

// We will use this as a floating reference so that we can set
// individual entries as we walk the table.
let mut v = &mut root.entries[vpn[2]];
// Now, we're going to traverse the page table and set the
// bits
// properly. We expect the root to be valid, however we're
// required to
// create anything beyond the root.
// In Rust, we create a range iterator using the .. operator.
// The .rev() will reverse the iteration since we need to
// start with
// VPN[2] The .. operator is inclusive on start but exclusive
// on end.
// So, (0..2) will iterate 0 and 1.
for i in (level..2).rev() {
    if !v.is_valid() {
        // Allocate a page
        let page = zalloc(1);
        // The page is already aligned by 4,096, so store it
        // directly The page is stored in the entry shifted
        // right by 2 places.
        v.set_entry(
            (page as i64 >> 2)
            | EntryBits::Valid.val(),
        );
    }
    let entry = ((v.get_entry() & !0x3ff) << 2) as *mut Entry;
    v = unsafe { entry.add(vpn[i]).as_mut().unwrap() };
}
// When we get here, we should be at VPN[0] and v should be

```

```

    ↪ pointing to
// our entry.
// The entry structure is Figure 4.18 in the RISC-V Privileged
// Specification
let entry = (ppn[2] << 28) as i64 | // PPN[2] = [53:28]
    (ppn[1] << 19) as i64 | // PPN[1] = [27:19]
    (ppn[0] << 10) as i64 | // PPN[0] = [18:10]
    bits | // Specified bits, such as User, Read, Write,
    ↪ etc
    EntryBits::Valid.val(); // Valid bit
    // Set the entry. V should be set to the correct
    ↪ pointer by the loop
    // above.
v.set_entry(entry);

```

在上面的代码中，我们用 `zalloc` 分配了一个新的页面。幸运的是，RISC-V 的每级页表正好是 4096 字节（512 个页表项，每个 8 字节），这正是我们使用 `zalloc` 分配的大小。

我的学生面临的一个挑战是，如果你看一下，PPN[2:0] 在 PTE 和物理地址中是一样的，只是它们的位置不同。由于某些原因，PPN[2:0] 被向右移了两位，而物理地址从第 10 位开始。而在物理地址中是从第 12 位开始的。然而，使用我上面采取的方法，即单独对 PPN 本身进行编码，我们应该不会有什么问题。

另一个有趣的方面是，无论我们在哪一级页表，我总是对 PPN[2:0] 进行编码。这对页表项来说还好，但它可能会造成混乱。记住，如果我们停在第 1 级，那么 PPN[0] 就不会被从 PTE 中复制出来。相反，PPN[0] 是从 VPN[0] 中复制的。这是由 MMU 硬件执行的。最后，我们只是添加了比页表项可能需要的更多的信息。

A.2.11 释放 MMU 表

我们将为每个用户空间的应用程序至少分配一个页。然而，我计划只允许用户空间的应用程序使用 4KiB 大小的页。因此，我们至少需要为一个地址准备三个页表项。这意味着，如果我们不重复使用这些页，内存很快就会耗尽。为了释放内存，我们将使用 `unmap`。

```

pub fn unmap(root: &mut Table) {
    // Start with level 2
    for lv2 in 0..Table::len() {
        let ref entry_lv2 = root.entries[lv2];
        if entry_lv2.is_valid() && entry_lv2.is_branch() {
            // This is a valid entry, so drill down and free.
            let memaddr_lv1 = (entry_lv2.get_entry() & !0x3ff) <<
            ↪ 2;
            let table_lv1 = unsafe {

```

```

        // Make table_lv1 a mutable reference instead of a
        // ↪ pointer.
        (memaddr_lv1 as *mut Table).as_mut().unwrap()
    };
    for lv1 in 0..Table::len() {
        let ref entry_lv1 = table_lv1.entries[lv1];
        if entry_lv1.is_valid() && entry_lv1.is_branch()
        {
            let memaddr_lv0 = (entry_lv1.get_entry()
                               & !0x3ff) << 2;
            // The next level is level 0, which
            // cannot have branches, therefore,
            // we free here.
            dealloc(memaddr_lv0 as *mut u8);
        }
    }
    dealloc(memaddr_lv1 as *mut u8);
}
}
}

```

就像 `map` 函数一样，这个函数假设了一个合法的页表根（第 2 级页表）。我们把它的可能引用传递到 `unmap` 函数中。因此，这段代码背后的逻辑是，我们从最低一级（第 0 级）开始释放空间，然后一路回到最高级（第 2 级）。上述代码本质上是使用迭代方式实现了一个递归函数。请注意，我没有释放页表根本身。由于我们得到了一个通用的 `Table` 结构体的引用，我们可以传入一个 1 级表来释放一个更大的表。

A.2.12 遍历页表

最后，我们需要手动遍历页表。这被用于从虚拟内存地址复制数据。由于虚拟内存地址在内核和用户进程之间是不同的，我们需要通过物理地址进行转化。唯一的方法是将虚拟内存地址转换为对应的物理地址。与硬件 MMU 不同，我们可以将任何页表传递给这个函数，无论这个表目前是否正被 MMU 使用。

```

pub fn virt_to_phys(root: &Table, vaddr: usize) -> Option {
    // Walk the page table pointed to by root
    let vpn = [
        // VPN[0] = vaddr[20:12]
        (vaddr >> 12) & 0x1ff,
        // VPN[1] = vaddr[29:21]
        (vaddr >> 21) & 0x1ff,
        // VPN[2] = vaddr[38:30]
        (vaddr >> 30) & 0x1ff,
    ];
}

```

```

let mut v = &root.entries[vpn[2]];
for i in (0..=2).rev() {
    if v.is_invalid() {
        // This is an invalid entry, page fault.
        break;
    }
    else if v.is_leaf() {
        // According to RISC-V, a leaf can be at any level.

        // The offset mask masks off the PPN. Each PPN is 9
        // bits and they start at bit #12. So, our formula
        // 12 + i * 9
        let off_mask = (1 << (12 + i * 9)) - 1;
        let vaddr_pgoff = vaddr & off_mask;
        let addr = ((v.get_entry() << 2) as usize) & !off_mask
            ↪ ;
        return Some(addr | vaddr_pgoff);
    }
    // Set v to the next entry which is pointed to by this
    // entry. However, the address was shifted right by 2
    ↪ places
    // when stored in the page table entry, so we shift it
    ↪ left
    // to get it back into place.
    let entry = ((v.get_entry() & !0x3ff) << 2) as *const
        ↪ Entry;
    // We do i - 1 here, however we should get None or Some
    ↪ () above
    // before we do 0 - 1 = -1.
    v = unsafe { entry.add(vpn[i - 1]).as_ref().unwrap() };
}

// If we get here, we've exhausted all valid tables and
    ↪ haven't
// found a leaf.
None
}

```

在上面的 Rust 函数中,我们得到了一个对页表的常引用。我们返回一个 Option, 它是一个枚举类型, 要么是 None (用于指示页面错误), 要么是 Some() (用于返回物理地址)。

使用 RISC-V 内存管理单元时有几种情况会产生页面错误。一个是遇到了一个无效的页表项 (V 位为 0)。另一个是在第 0 级有枝干项。由于 0 级是最低的级别, 如果我们发现一个更低的级别 (比如 -1?), 就会发生页面故障。

A.2.13 A 和 D 位

如果你读过 RISC-V 的特权规范，D (dirty) 和 A (accessed) 位可以有两种不同的含义。

对于应用处理器来说，更传统的含义是，当内存被读取或写入时，CPU 都会将 A 位设置为 1，而内存被写入时 CPU 会将 D 位设置为 1。然而，RISC-V 规范本质上也允许硬件将这些作为冗余的 R 和 W 位—至少我是这么理解的。换句话说，由我们内核程序员来控制 A 和 D 位，而不是硬件。

HiFive Unleashed 开发板使用的是后者，即如果 D 位（用于存储）或 A 位（用于加载）在页表项中为 0，MMU 会抛出一个页面错误，所以让我们来仔细看看。

1. 如果你试图写入一个 D 位为 0 的页面，MMU 将抛出一个页面错误，这与 W 位为 0 的情况很相似。2. 如果你试图读取一个 A 位为 0 的页面，MMU 将抛出一个页面错误，这与 R 位为 0 的情况很相似。

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are permitted:

- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. The PTE update must be exact (i.e., not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

The PTE update is not required to be atomic with respect to the explicit memory access that caused the update, and the sequence is interruptible. However, the hart must not perform the explicit memory access before the PTE update is globally visible.

All harts in a system must employ the same PTE-update scheme as each other.

图 A.6 SV39 MMU A 和 D 位

图 A.6 出自 RISC-V 特权规范^[1]，第 4.3.1 章。

简而言之，我们有两种选择：(a) 软件控制或 (b) 硬件控制。

A.2.14 内核的地址映射

我们将切换到内核态，这样可以让内核内存使用 MMU，不过我们首先需要映射一切我们需要的内容。在 RISC-V 中，机器态使用物理内存地址；而如果我们打

开 MMU (将 `MODE` 字段设置为 8), 那么我们可以在内核或用户态下使用 MMU。

要做到这一点, 我们要对内核中需要的一切进行恒等映射 (虚拟地址 = 物理地址), 包括程序代码、全局段、UART MMIO 地址等等。首先, 我用 Rust 写了一个函数, 它将帮助我恒等映射一段地址。

```
pub fn id_map_range(root: &mut page::Table,
    start: usize,
    end: usize,
    bits: i64)
{
    let mut memaddr = start & !(page::PAGE_SIZE - 1);
    let num_kb_pages = (page::align_val(end, 12)
        - memaddr)
        / page::PAGE_SIZE;

    // I named this num_kb_pages for future expansion when
    // I decide to allow for GiB (2^30) and 2MiB (2^21) page
    // sizes. However, the overlapping memory regions are
    // ↪ causing
    // nightmares.
    for _ in 0..num_kb_pages {
        page::map(root, memaddr, memaddr, bits, 0);
        memaddr += 1 << 12;
    }
}
```

如你所见, 这个函数调用了我们的 `page::map` 函数, 它将一个虚拟地址映射到一个物理地址。我们得到的根表是一个可变引用, 可以把它直接传递给 `map` 函数。

现在, 我们需要用这个函数来映射我们的程序代码段和稍后设备驱动所需的所有的 MMIO 地址。

```
page::init();
kmem::init();

// Map heap allocations
let root_ptr = kmem::get_page_table();
let root_u = root_ptr as usize;
let mut root = unsafe { root_ptr.as_mut().unwrap() };
let kheap_head = kmem::get_head() as usize;
let total_pages = kmem::get_num_allocations();
println!();
println!();
unsafe {
    println!("TEXT: 0x{:x} -> 0x{:x}", TEXT_START, TEXT_END);
    println!("RODATA: 0x{:x} -> 0x{:x}", RODATA_START,
        ↪ RODATA_END);
    println!("DATA: 0x{:x} -> 0x{:x}", DATA_START, DATA_END);
}
```

```

println!("BSS:    0x{:x} -> 0x{:x}", BSS_START, BSS_END);
println!("STACK: 0x{:x} -> 0x{:x}", KERNEL_STACK_START,
    ↪ KERNEL_STACK_END);
println!("HEAP:   0x{:x} -> 0x{:x}", kheap_head, kheap_head
    ↪ + total_pages * 4096);
}
id_map_range(
    &mut root,
    kheap_head,
    kheap_head + total_pages * 4096,
    page::EntryBits::ReadWrite.val(),
);
unsafe {
    // Map heap descriptors
    let num_pages = HEAP_SIZE / page::PAGE_SIZE;
    id_map_range(&mut root,
        HEAP_START,
        HEAP_START + num_pages,
        page::EntryBits::ReadWrite.val()
    );
    // Map executable section
    id_map_range(
        &mut root,
        TEXT_START,
        TEXT_END,
        page::EntryBits::ReadExecute.val(),
    );
    // Map rodata section
    // We put the Rodata section into the text section, so they
    ↪ can
    // potentially overlap however, we only care that it's read
    // only.
    id_map_range(
        &mut root,
        RODATA_START,
        RODATA_END,
        page::EntryBits::ReadExecute.val(),
    );
    // Map data section
    id_map_range(
        &mut root,
        DATA_START,
        DATA_END,
        page::EntryBits::ReadWrite.val(),
    );
    // Map bss section
    id_map_range(
        &mut root,
        BSS_START,
        BSS_END,
        page::EntryBits::ReadWrite.val(),
    );
}

```

```

);
// Map kernel stack
id_map_range(
    &mut root,
    KERNEL_STACK_START,
    KERNEL_STACK_END,
    page::EntryBits::ReadWrite.val(),
);
}

// UART
page::map(
    &mut root,
    0x1000_0000,
    0x1000_0000,
    page::EntryBits::ReadWrite.val(),
    0
);

```

这有很多代码，因此我需要编写 `id_map_range` 函数。在这段代码中，我们首先初始化我们在第 3.1 章中创建的页分配系统。然后，我们创建一个页表根，它正好需要 1 页（4096 字节）。来回转换内存地址的类型的意义在于，我们最终需要将一个物理地址写入 SATP（Supervisor Address Translation and Protection，监管者地址翻译与保护）寄存器以使 MMU 正常运行。

现在我们有页表根，我们需要把它写进 SATP 寄存器的 PPN 字段。PPN 本质上是页表根的 56 位物理地址的前 44 位。我们要做的是把页表根的地址向右移 12 位，这本质上是把地址除以一个页面的大小。

```

let root_ppn = root_u >> 12;
let satp_val = 8 << 60 | root_ppn;
unsafe {
    asm!("csw satp, $0" :: "r"(satp_val));
}

```

上面的代码使用了 `asm!` 宏，它允许我们在 Rust 中直接编写汇编代码。在这里，我们使用 `csw` 指令，意思是“控制和状态寄存器-写入”。`8 << 60` 把数值 8 放在 SATP 寄存器的 MODE 字段中，表示 Sv39 分页模式。

A.2.15 开启 MMU

仅在 MMU 模式字段中写入 8 并没有完全打开 MMU。原因是我们处于 3 号 CPU 模式，也就是机器模式。所以，我们需要切换到“监督者”模式，也就是模式 1。我们通过在 `mstatus` 寄存器的 MPP（Machine Previous Privilege）字段（第 11 和 12 位）中写入二进制数字 01 来实现这一点。

我们切换 SATP 寄存器时需要谨慎，因为页表会被缓存。切换的次数越多，我们就需要刷新缓存：旧的出来，新的进去。当用户进程开始切入和切出上下文时，我们需要同步页表。如果你现在对此有兴趣，请看 RISC-V 的特权规范^[1] 第 4.2.1 章，不过当我们开始运行进程时，我们会详细介绍这一点。

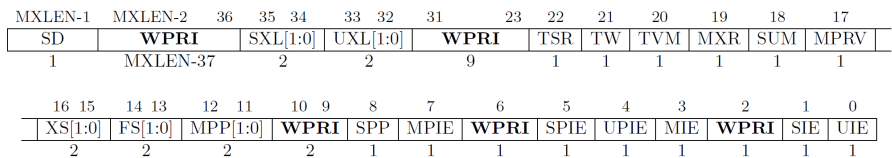


Figure 3.7: Machine-mode status register (`mstatus`) for RV64.

图 A.7 `mstatus` 寄存器

```
li t0, (1 << 11) | (1 << 5)
csrw mstatus, t0

la t1, kmain
csrw mepc, t1

mret
```

我们在上面的代码中所做的是启用中断，并将第 11 位开始的数值置为 1 (`MPP=01`)。当我们执行 `mret` 时，我们进入了一个叫 `kmain` 的 Rust 函数。现在我们处于监督者模式并且完全启用了 MMU。

我们还没有处理任何页面错误，所以要小心！如果你跑到地址空间以外的地方，你的内核就会挂起宕机。

A.3 处理中断和陷入

A.3.1 概述

如果我们从来不会出差错，那不是很好吗？嗯，很不好，这是不可能的。所以，我们需要在问题发生时做好准备。CPU 能够在事情发生时通知内核。虽说如此，并非所有的通知都是坏事。比如系统调用，或者定时器中断呢？是的，这些也会导致 CPU 通知内核。

A.3.2 RISC-V 中断系统

RISC-V 系统使用一个单一的、指向内核中的物理地址的函数指针。每当有事情发生时，CPU 会切换到机器态并跳转到这个函数。在 RISC-V 中，我们有两个特

殊的 CSR（控制和状态寄存器）来控制这种 CPU 通信。

第一个寄存器是 `mtvec` 寄存器，它代表了机器陷入向量。向量是一个函数指针。每当有事情发生，CPU 就会“调用”这个寄存器所表示的函数。

3.1.7 Machine Trap-Vector Base-Address Register (`mtvec`)

The `mtvec` register is an MXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

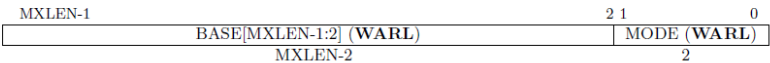


Figure 3.8: Machine trap-vector base-address register (`mtvec`).

The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

Value	Name	Description
0	Direct	All exceptions set <code>pc</code> to BASE.
1	Vectored	Asynchronous interrupts set <code>pc</code> to BASE+4×cause.
≥2	—	Reserved

Table 3.5: Encoding of `mtvec` MODE field.

图 A.8 `mtvec` 寄存器

`mtvec` 寄存器有两个不同的字段。BASE，也就是函数的地址，以及 MODE，它决定了我们是要使用直接中断还是向量中断。我们要使用 Rust 的 `match` 来重定向陷入。所以我们需要确保最后两位是 0，这意味着我们的函数地址需要是 4 的倍数。

使用直接模式的 `mtvec` 意味着所有的陷入都会进入完全相同的函数，而使用矢量模式的 `mtvec` 会根据陷入原因进入不同的函数。简单起见，我将使用直接模式。然后，我们可以使用 Rust 的 `match` 语句解析出陷入原因。

A.3.3 为什么陷入？

`mcause` 寄存器（机器态陷入原因）会给你一个异常代码，概括地解释是什么导致了陷入。陷入有两种不同的类型：异步的和同步的。异步陷入意味着是当前执行的指令之外的东西引起了 CPU 的“陷入”。同步陷入意味着是当前执行的指令引起了“陷入”。异步陷入的原因代码最高位一定为 1。同步陷入的原因代码最高位一定为 0。

1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	<i>Reserved for future standard use</i>
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	<i>Reserved for future standard use</i>
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	<i>Reserved for future standard use</i>
1	11	Machine external interrupt
1	12–15	<i>Reserved for future standard use</i>
1	≥ 16	<i>Reserved for platform use</i>

图 A.9 mcause 异步陷入编码

0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	<i>Reserved</i>
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved for future standard use</i>
0	15	Store/AMO page fault
0	16–23	<i>Reserved for future standard use</i>
0	24–31	<i>Reserved for custom use</i>
0	32–47	<i>Reserved for future standard use</i>
0	48–63	<i>Reserved for custom use</i>
0	≥ 64	<i>Reserved for future standard use</i>

图 A.10 mcause 同步陷入编码

A.3.4 简单启动

我对启动代码进行了修改以简化启动过程。我们通过 `kinit` 进入 **Rust**，它在仅有物理内存的机器态下运行代码。在这个函数中，我们基本上可以自任意支配我们的系统。然而，`kinit` 的意义在于让我们尽快进入 `kmain`。我们将在内核态下运行 `kmain` 函数，在这种模式下，我们的内核开启了虚拟内存。

```
# We need a stack for our kernel. This symbol comes from virt.
↪ lds
la sp, _stack_end

# Setting \lstinline{mstatus} register:
# 0b01 << 11: Machine's previous protection mode is 2 (MPP=2).
li t0, 0b11 << 11
csrw mstatus, t0

# Do not allow interrupts while running kinit
csrw mie, zero

# Machine's exception program counter (MEPC) is set to \
↪ lstinline{kinit\lstinline{.
la t1, kinit
csrw mepc, t1

# Set the return address to get us into supervisor mode
la ra, 2f

# We use mret here so that the mstatus register is properly
↪ updated.
mret
```

我们的代码设置好了机器态环境。我们在 `mie` 中写入 0，也就是“机器态中断使能寄存器”，以禁用所有的中断。因此，`kinit` 处于机器态，只有物理内存，并且现在是不可抢占的。这使我们能够设置我们的机器，而不必担心其他的硬件线程（核）会干扰我们的进程。

A.3.5 陷入

陷入本质上是 CPU 通知内核的一种方式。通常情况下，我们作为内核程序员告诉 CPU 正在发生什么。然而，有时 CPU 需要让我们知道发生了什么。如上所述，它是通过一个陷入来实现的。所以，为了在 **Rust** 中处理陷入，我们需要创建一个新的文件，`trap.rs`。

```
// trap.rs
// Trap routines
// Stephen Marz
```



```

// 10 October 2019

use crate::cpu::TrapFrame;

#[no_mangle]
extern "C" fn m_trap(epc: usize,
                    tval: usize,
                    cause: usize,
                    hart: usize,
                    status: usize,
                    frame: &mut TrapFrame)
    -> usize
{
    // We're going to handle all traps in machine mode. RISC-V
    ↪ lets
    // us delegate to supervisor mode, but switching out SATP (
    ↪ virtual memory)
    // gets hairy.
    let is_async = {
        if cause >> 63 & 1 == 1 {
            true
        }
        else {
            false
        }
    };
    // The cause contains the type of trap (sync, async) as
    ↪ well as the cause
    // number. So, here we narrow down just the cause number.
    let cause_num = cause & 0xffff;
    let mut return_pc = epc;
    if is_async {
        // Asynchronous trap
        match cause_num {
            3 => {
                // Machine software
                println!("Machine software interrupt CPU#{}", hart
                    ↪ );
            },
            7 => unsafe {
                // Machine timer
                let mtimecmp = 0x0200_4000 as *mut u64;
                let mtime = 0x0200_bff8 as *const u64;
                // The frequency given by QEMU is 10_000_000 Hz,
                ↪ so this sets
                // the next interrupt to fire one second from now.
                mtimecmp.write_volatile(mtime.read_volatile() + 10
                    ↪ _000_000);
            },
            11 => {
                // Machine external (interrupt from Platform

```

```

        ↪ Interrupt Controller (PLIC))
        println!("Machine external interrupt CPU#{}", hart
        ↪ );
    },
    _ => {
        panic!("Unhandled async trap CPU#{} -> {}\n", hart
        ↪ , cause_num);
    }
}
}
else {
    // Synchronous trap
    match cause_num {
        2 => {
            // Illegal instruction
            panic!("Illegal instruction CPU#{} -> 0x{:08x}: 0x
            ↪ {:08x}\n", hart, epc, tval);
        },
        8 => {
            // Environment (system) call from User mode
            println!("E-call from User mode! CPU#{} -> 0x{:08x}
            ↪ ", hart, epc);
            return_pc += 4;
        },
        9 => {
            // Environment (system) call from Supervisor mode
            println!("E-call from Supervisor mode! CPU#{} -> 0
            ↪ x{:08x}", hart, epc);
            return_pc += 4;
        },
        11 => {
            // Environment (system) call from Machine mode
            panic!("E-call from Machine mode! CPU#{} -> 0x{:08
            ↪ x}\n", hart, epc);
        },
        // Page faults
        12 => {
            // Instruction page fault
            println!("Instruction page fault CPU#{} -> 0x{:08x}
            ↪ : 0x{:08x}", hart, epc, tval);
            return_pc += 4;
        },
        13 => {
            // Load page fault
            println!("Load page fault CPU#{} -> 0x{:08x}: 0x
            ↪ {:08x}", hart, epc, tval);
            return_pc += 4;
        },
        15 => {
            // Store page fault
            println!("Store page fault CPU#{} -> 0x{:08x}: 0x

```

```

        ↪ {:08x}", hart, epc, tval);
    return_pc += 4;
},
_ => {
    panic!("Unhandled sync trap CPU#{} -> {}\n", hart,
        ↪ cause_num);
}
}
};
// Finally, return the updated program counter
return_pc
}

```

上面的代码就是我们在陷入时跳转到的函数。CPU 遇到一个陷入，找到 `mtvec`，然后跳转到其中指定的地址。注意，我们必须先解析出中断类型（异步或同步），然后再解析其原因。

请注意，我们三个期望的同步异常：8、9 和 11。这些是“环境”调用，也就是系统调用。然而，原因是根据我们在发出 `ecall` 指令时所处的模式来解析的。我对机器态的 `ecall` 调用了 `panic!`，因为我们永远不应该遇到一个机器态的 `ecall`。相反，我们在内核态下运行内核，而用户应用程序将通过用户态的 `ecall` 进入内核。

当我们遇到一个陷入时，我们需要跟踪我们的寄存器等等。我们如果在一开始就搞乱了寄存器，就没有办法再重新启动一个用户应用。我们将把这些信息保存到一个叫做陷入帧的东西中。我们还将使用 `mscratch` 寄存器来存储这些信息，这样当我们遇到陷入的时候就很容易找到它。

A.3.6 陷入帧 Rust 结构体

```

#[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
}

```

如上述结构体所示，我们存储了所有的通用寄存器、浮点寄存器、SATP(MMU)、处理陷入的堆栈，以及硬件线程 ID。当我们只使用机器态的陷入时，最后一项是不必要的。RISC-V 允许我们将某些陷入委托给内核态。然而，我们还没有这样做。目前 `hartid` 是多余的，因为我们可以通过 `csrr a0, mhartid` 获得硬件线程 ID。

你还会注意到两个 Rust 指令：`#[repr(C)]` 和 `#[derive(Clone, Copy)]`。

第一个指令使我们的结构体遵循 C 语言风格的结构体。这么做是因为当我们在汇编中操作陷入帧时，我们需要知道每个成员变量的偏移量。最后，derive 指令将使 Rust 自动实现 Copy 和 Clone 的特性。如果我们不使用 derive，我们就得自己实现。

现在我们已经建立起了陷入帧，让我们看看它在汇编中的样子。

A.3.7 查看汇编中的陷入向量

```
.option norvc
m_trap_vector:
# All registers are volatile here, we need to save them
# before we do anything.
csrrw t6, mscratch, t6
# csrrw will atomically swap t6 into mscratch and the old
# value of mscratch into t6. This is nice because we just
# switched values and didn't destroy anything -- all
    ↪ atomically!
# in cpu.rs we have a structure of:
# 32 gp regs 0
# 32 fp regs 256
# SATP register 512
# Trap stack 520
# CPU HARTID 528
# We use t6 as the temporary register because it is the very
# bottom register (x31)
.set i, 1
.rept 30
    save_gp %i
    .set i, i+1
.endr

# Save the actual t6 register, which we swapped into
# mscratch
mv t5, t6
csrr t6, mscratch
save_gp 31, t5

# Restore the kernel trap frame into mscratch
csrw mscratch, t5

# Get ready to go into Rust (trap.rs)
# We don't want to write into the user's stack or whomever
# messed with us here.
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

# When we get here, we've returned from m_trap, restore
# ↪ registers
# and return.
# m_trap will return the return address via a0.

csrw mepc, a0

# Now load the trap frame back into t6
csrr t6, mscratch

# Restore all GP registers
.set i, 1
.rept 31
    load_gp %i
    .set i, i+1
.endr

# Since we ran this loop 31 times starting with i = 1,
# the last one loaded t6 back to its original value.

mret

```

你可以看到我们使用了所谓的汇编器指令和宏，例如 `.set` 和 `store_gp`，这使我们的生活更轻松。这本质上是一个汇编时的循环，当我们汇编这个文件时会被展开。

我们还指定了 `.option norvc`，意思是“RISC-V 压缩指令”，它是 RISC-V ISA 的 C 扩展。这强制要求所有用于陷入向量的指令都是 32 位的。这不是特别重要，但是当我们添加多个陷入向量时，我们需要确保每个向量函数都从 4 的整数倍的内存地址开始。这是因为 `mtvec` 寄存器使用最后两位在直接模式和向量模式间切换。

A.3.8 GNU 汇编宏

宏的定义如下。

```

.altmacro
.set NUM_GP_REGS, 32 # Number of registers per context
.set NUM_FP_REGS, 32
.set REG_SIZE, 8 # Register size (in bytes)
.set MAX_CPUS, 8 # Maximum number of CPUs

# Use macros for saving and restoring multiple registers
.macro save_gp i, basereg=t6

```

```

    sd x\i, ((\i)*REG_SIZE)(\basereg)
.endm
.macro load_gp i, basereg=t6
    ld x\i, ((\i)*REG_SIZE)(\basereg)
.endm
.macro save_fp i, basereg=t6
    fsd f\i, ((NUM_GP_REGS+(\i))*REG_SIZE)(\basereg)
.endm
.macro load_fp i, basereg=t6
    fld f\i, ((NUM_GP_REGS+(\i))*REG_SIZE)(\basereg)
.endm

```

讲解上面的宏已经超出了本博客的范围，但值得一提的是，我们正在执行以下指令之一：sd, ld, fsd, fld，并且默认情况下，我们使用 t6 寄存器。我选择 t6 寄存器的原因是它是 31 号寄存器，这使得我们很容易追踪到我们 从寄存器 0 一直运行到 31 的循环。

A.3.9 这有什么作用？

陷入是由 CPU 引起的。每个硬件线程都可以产生陷入，所以我们之后需要进行互斥和推迟行动。

当你运行这段代码时，应该看到如图 A.11 所示的输出。

```

Walk 0x80262fff = 0x80262fff
Setting 0x80000000000080259
Scratch reg = 0x800470d0
Boxed value = 100
String = 💖

Allocations of a box, vector, and string
0x80059000: Length = 16          Taken = true
0x80059010: Length = 16          Taken = true
0x80059020: Length = 2097120     Taken = false

Everything should now be free:
0x80059000: Length = 2097152     Taken = false
Store page fault CPU#0 -> 0x80000d8e: 0x00000000

```

图 A.11 第四章输出结果

你会注意到，在底部有一个页面错误。这是有意为之，因为我已经在 Rust 代码中加入了以下内容。这几乎确保了页面错误的发生！

```

unsafe {

```

```

// Set the next machine timer to fire.
let mtimecmp = 0x0200_4000 as *mut u64;
let mtime = 0x0200_bff8 as *const u64;
// The frequency given by QEMU is 10_000_000 Hz, so this
    ↪ sets
// the next interrupt to fire one second from now.
mtimecmp.write_volatile(mtime.read_volatile() + 10_000_000)
    ↪ ;

// Let's cause a page fault and see what happens. This
    ↪ should trap
// to m_trap under trap.rs
let v = 0x0 as *mut u64;
v.write_volatile(0);
}

```

第一部分重置了 CLINT 定时器,这将触发一个异步机器定时器陷入。然后,我们对 NULL 指针解引用,这将导致我们的存储页面错误。如果是 `v.read_volatile` ↪ `()`, 我们会得到一个载入页面错误而非存储。

参考文献

- [1] Andrew Waterman, Krste Asanović, and John Hauser, editors. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. RISC-V International, December 2021.

书面翻译对应的原文索引

- [1] Stephen Marz. *The adventures of os: Making a risc-v operating system using rust*, 2019.