

摘要

实时操作系统具有实时性、便携性、低功耗等优点，可以满足各种不同应用的需求，广泛应用于各领域。但是大部分实时操作系统和处理器的核心技术被国外垄断，因此处理器和实时操作系统成为了急需解决的“卡脖子”关键技术。RISC-V 架构作为一个开源指令架构，是我国处理器芯片自控可控设计的首选。根据实际应用需求迫切需要一款基于 RISC-V 架构的实时操作系统。本文完成了基于 RISC-V 架构的实时操作系统 RVOS 的设计与应用测试。

论文分析了实时操作系统的基本结构和设计思路，并对 RISC-V 指令集中非特权指令、特权指令和 Trap 处理流程进行了详细分析。根据 RISC-V 的结构特点，提出基于 RISC-V 的实时操作系统 RVOS 的设计方案，包括串口输出、内存管理、任务调度和 Trap 处理等功能模块。根据 RVOS 的整体功能结构，进行系统各个功能模块设计，并在 64 位的 RISC-V 处理器芯片 K210 上完成了系统移植。论文在 QEMU-Virt 模拟平台与 K210 芯片上完成了所设计操作系统 RVOS 的各个功能模块及整体系统的测试验证。实验结构表明所设计操作系统功能正确性。

本文所设计的 RVOS 可以完整执行实时操作系统的基本功能，其轻量化的结构可以适用小型化嵌入式设备。论文工作对基于 RISC-V 架构的实时操作系统设计具有一定的参考意义和实用价值。

关键词：RISC-V 实时操作系统 K210

ABSTRACT

The real-time operating system has the advantages of real-time, portability and low power consumption, which can meet the needs of various applications and is widely used in various fields. However, most of the core technologies of real-time operating system and processor are monopolized by foreign countries, so processor and real-time operating system become the key technologies that need to be solved urgently. RISC-V architecture, as an open source instruction architecture, is the first choice for the control design of processor chips in our country. A real - time operating system based on RISC-V architecture is urgently needed. This paper completes the design and application test of RVOS, a real-time operating system based on RISC-V architecture.

This paper analyzes the basic structure and design idea of real-time operating system, and analyzes the non-privileged instruction, privileged instruction and Trap processing flow of RISC-V instruction set in detail. According to the structure characteristics of RISC-V, the design scheme of RVOS based on RISC-V is proposed, including serial port output, memory management, task scheduling, Trap processing and other functional modules. According to the overall functional structure of RVOS, each functional module of the system is designed, and the system transplantation is completed on the 64-bit RISC-V processor chip K210. In this paper, the functional modules and the whole system of the designed operating system RVOS are tested and verified on the QEMU-Virt simulation platform and K210 chip. The experimental structure shows that the designed operating system is functional correct.

The RVOS designed in this paper can perform the basic functions of real-time operating system completely, and its lightweight structure is suitable for miniaturized embedded devices. The work of this paper has certain reference significance and practical value to the design of real-time operating system based on RISC-V architecture.

Key words: RISC-V Real-time operating system K210

目 录

第一章 绪论	1
1.1 研究目的和意义	1
1.2 国内外研究现状	2
1.3 论文组织结构	3
第二章 基于 RISC-V 架构的实时操作系统分析	5
2.1 实时操作系统	5
2.2 RISC-V 指令集	7
2.2.1 RISC-V 非特权指令集	8
2.2.2 RISC-V 特权指令集	11
2.2.3 RISC-V Trap 处理流程	13
2.3 K210 芯片简介	15
第三章 RVOS 的设计与移植方案	17
3.1 RVOS 初始化与 UART 输出	17
3.2 内存管理模块	18
3.3 协作式任务调度	20
3.4 Trap 处理	21
3.4.1 异常处理	22
3.4.2 中断处理	22
3.5 RVOS 的 64 位扩展	24
3.6 基于 K210 的 RVOS 移植	25
第四章 RVOS 各模块设计细节	27
4.1 系统初始化	27
4.2 内存管理模块	28
4.3 任务调度器	31
4.4 Trap 处理	35
第五章 RVOS 功能模块测试	37
5.1 测试环境搭建	37

5.2 RVOS 初始化测试	38
5.3 内存管理测试.....	40
5.3.1 page 级内存管理	40
5.3.2 malloc 级内存管理.....	42
5.4 任务调度模块测试.....	44
5.5 Trap 处理测试	46
5.5.1 异常测试.....	46
5.5.2 中断测试.....	46
第六章 总结与展望	48
致 谢.....	49
参考文献.....	50

第一章 绪论

1.1 研究目的和意义

RISC-V 是一种新兴的开源精简指令集架构。由加州大学伯克利分校在 2010 年首次发布^[1]。RISC-V 的出现和迅速发展有其必然的原因，它是建立在现有的体系结构(如 x86、ARM、MIPS 等)经长期发展所暴露出的种种问题之上。顺应现代信息系统设计需求和体系结构发展趋势而生的。

(1) 现有的各类体系结构往往缺乏开放性，存在许多知识产权、政治干预等非技术性问题。例如，Intel 公司从 1978 年开始一直垄断 x86 架构的专利。其它公司在使用 x86 指令集相关技术时需要向其支付高昂的授权费用。对 x86 指令集的模拟也会引发法律上的争议^[2]。这种封闭的态势与体系结构发展的开放趋势背道而驰，抬高了系统研发与成果转化的成本，阻碍了技术的推广和进步。RISC-V 作为一种新兴体系结构，具有开源、免费、开放、自由的特点，其基金会总部于 2020 年 3 月正式迁往永久中立国瑞士^[3]，更是释放了坚持服务全世界的信号，使任何组织和个人都可以不受地缘政治影响、自由平等地使用 RISC-V。

(2) 现有体系结构经过了许多版本的迭代，有非常多的历史遗留问题。由于旧版本与新版本的技术产品在市场生态中共存，导致新版本的研发必须考虑兼容性问题，来支持一些过时的定义和其实用不到的技术特性。例如，AMD64 是对 32 位 x86 架构的 64 位扩展，面向 64 位开发与应用环境；但它同时仍要向前兼容 32 位甚至 16 位的 x86 架构，使早期 x86 架构下开发的应用同样可以在 AMD64 系统中正常运行。这种积重难返的状态削弱了现有体系结构的可定制化能力，难以满足现代信息系统对于多样化的工作环境与功能表现的需求。RISC-V 作为一种从零开始设计的新体系结构，吸收了现有各体系结构优点的同时，去除了对历史遗留问题的顾及和旧有技术的依赖；进一步地，RISC-V 采用模块化设计，并提供大量自定义编码空间以支持对指令集的扩展，这允许了开发者根据资源、能耗、权限、实时性等不同需求，基于部分特定的模块和扩展指令集进行精细化的系统设计研发，体现了强大的系统可定制化能力。

(3) 现有主流架构的文档资源种类繁多、内容冗长，学习与维护的成本较高，使开发者难以在短时间内掌握所需的技术，遇到问题时也不易迅速定位到相

关的信息区间。例如，ARMv8-A 架构的官方手册^[4]仅一卷就多达 8 538 页；相比之下，RISC-V 官方手册仅有两卷 329 页，包括 238 页的指令集手册^[5]和 91 页的特权架构手册，文档精简，学习门槛更低，更有助于研发团队的不断壮大和技术实力的不断进步。

结合当前中美之间的科技战，RISC-V 架构能完美的承接起对主流指令集（x86，ARM）的替代工作，在国防安全上也具有重大意义。随着 RISC-V 在国内外的逐渐推广，对于小型的 RISC-V 嵌入式设备，对实用的 RTOS 的需求必然扩大，本毕设就是针对该方向的需求进行研究。

1.2 国内外研究现状

（一）国内外研究现状

对于 RISC-V 架构的 RTOS，目前国内外已有多家公司或教育机构给出相关产品，比较出名的如下：

1、FreeRTOS

- 作为老牌 RTOS。FreeRTOS^[6]是一个很流行的应用在嵌入式设备上的实时操作系统内核，诞生于 2003 年，采用 MIT 许可证发布，它具有以下特点：
- 设计小巧，整个代码只有 3 到 4 个 C 文件。
- 可读性强，易维护，大部分的代码都是 C 语言编写，很少的部分采用汇编语言。
- 支持优先级多线程、互斥锁、信号量和软件计时器，支持低功耗处理以及一定程度上的内存保护。
- 已经被移植到 K210,K510 微处理器上。

2、RT-Thread

RT-Thread^[7]是一个集实时操作系统（RTOS）内核、中间件组件和开发者社区于一体的技术平台，也是一个组件完整丰富、高度可伸缩、简易开发、超低功耗、高安全性的物联网操作系统。诞生于 2006 年。采用 Apache2.0 许可证发布。

它具有以下特点：

- 面向对象的实时内核。
- 8, 32 或 256 个优先级的多线程调度。对于同优先级线程使用时间片轮转调度法。

- 提供信号量，也提供互斥信号量以防止优先级反转。
- 支持其他高效通信方式，比如邮箱、消息队列和事件标志。
- 支持静态内存分配方法，也支持线程安全的动态内存管理。
- 对高层应用提供设备框架。
- 几乎支持市场上所有主流的 MCU 的 Wi-Fi 芯片。

对于 RISC-V 平台的 RTOS 移植情况，目前已有许多研究单位在不同的 RISC-V 平台上进行了移植，并且已经取得了一定的成果。其中较为著名的研究单位包括美国加州大学伯克利分校、中国科学院计算技术研究所等。这些研究单位已经在 RV32IMAC 架构上完成了 FreeRTOS 的移植，同时在 QEMU 模拟器和 HiFive1 RISC-V 开发板上进行了测试。同时，Zephyr RTOS 也已经被成功地移植到 RISC-V 平台上，并支持 RV32I 和 RV64G 架构。

（二）发展趋势

从当前的形势来看，RISC-V 目前最大的短板在于基于这一架构的生态发展还处于初级阶段，目前国内正在完善这张“拼图”的玩家尚有一些，如赛昉（SiFive）中国、芯来科技、平头哥、台湾晶心科技（Andes）等，但整体而言生态发展还在起步阶段^[8]。根据 Tractica 的预测，基于 RISC-V 的 IP 和软件工具的全球收入将在 2025 年增加到 11 亿美元，高于 2018 年的 5200 万美元。由此可见，对 RISC-V 架构的生态支持研发未来将是一个巨大的蓝海市场。而在这个市场中，基于 RISC-V 架构的 RTOS 也必将占用一定的份额。据了解，RISC-V 被认为适合应用在 IoT 市场。IoT 的相关设备通常内存较小且需要高度定制，此时一款合适的支持灵活定制的小型模块化 RTOS 必将备受青睐。

1.3 论文组织结构

本文的主要结构如下：

第一章为绪论，介绍了 RISC-V 指令集架构和实时操作系统发展的现状及本文的主要工作。

第二章引入与本文相关的实时操作系统概念以及 RISC-V 架构理论，同时介绍了 K210 芯片的特点。

第三章介绍了 RVOS 的操作系统设计思想以及移植方案。

第四章讲解了 RVOS 的详细实现原理以及移植方案的实际操作。

第五章展示了 RVOS 以及移植过程的相关源码以及必要的设计。

第六章对论文所完成工作进行总结，并对下一步工作进行展望。

第二章 基于 RISC-V 架构的实时操作系统分析

本文主要介绍实时操作系统(RTOS)及 RISC-V 架构相关技术。同时介绍 K210 芯片相关技术特点。

2.1 实时操作系统

实时操作系统(Real-time operating system, RTOS)^[9], 又称即时操作系统, 它会按照排序运行、管理系统资源, 并为开发应用程序提供一致的基础。与一般的操作系统相比, 实时操作系统最大的特色就是“实时性”, 如果有一个任务需要执行, 实时操作系统会马上(在较短时间内)执行该任务, 不会有较长的延时。这种特性保证了各个任务的及时执行。

实时操作系统通常与嵌入式操作系统一并出现, 但实际上这两种东西完全不同。虽然大多数实时操作系统都是嵌入式操作系统, 但嵌入式操作系统并不全都是实时的。民间对实时操作系统有一些常见的误区, 比如: 速度快, 吞吐量大, 代码精简, 代码规模小等等。其实这些都不算是实时操作系统的特性, 别的操作系统也可以做到。只有“实时性”才是 RTOS 的最大特征, 其它的都不算是。

实时运算(Real-time computing)^[10]是计算机科学中对受到“实时约束”的计算机硬件和计算机软件系统的研究, 实时约束像是从事件发生到系统回应之间的最长时间限制。实时程序必须保证在严格的时间限制内响应。所以, 每一个实时操作系统中都要包含一个实时任务调度器, 这个任务调度器与其它操作系统的最大不同是强调: 严格按照优先级来分配 CPU 时间, 并且时间片轮转不是实时调度器的一个必选项。

提出实时操作系统的概念, 可以至少解决两个问题, 首先是早期的 CPU 任务切换的开销太大, 实时调度器可以避免任务频繁切换导致 CPU 时间的浪费。其次在一些特殊的应用场景中, 必须要保证重要的任务优先被执行。在这样的背景下, 实时操作系统就被设计出来了, 典型的实时操作系统有 VxWorks, RT-Thread, uCOS, QNX, WinCE 等。

实时任务调度器^[11]是实时操作系统的一个必选项, 但不代表只要设计出来一个实时调度器就足够了。事实上设计一个实时调度内核并不是一个多么复杂的任务, 实时操作系统的特性是在整个操作系统的设计思路都要时刻关注实时性。

这些设计思路包括：

1. 实时的消息、事件处理机制。

常规的操作系统中，消息队列都是按照 FIFO（先进先出）^[12]的方式进行调度，如果有多个接受者，那么接受者也是按照 FIFO 的原则接受消息（数据），但实时操作系统会提供基于优先级的处理方式：两个任务优先级是分别是 10 和 20，同时等待一个信号量，如果按照优先级方式处理，则优先级为 10 的任务会优先收到信号量。

2. 提供内核级的优先级翻转处理方式

实时操作系统调度器最经常遇到的问题就是优先级翻转，因此对于类似信号量一类的 API，都能提供抑止优先级翻转的机制，防止操作系统死锁。

3. 减少粗粒度的锁和长期关中断的使用

这里的锁主要是指自旋锁(spinlock) ^[13]一类会影响中断的锁，也包括任何关中断的操作。在 Windows 和 Linux 的驱动中，为了同步的需要，可能会长期关闭中断，这里的长期可能是毫秒到百微秒级。但实时操作系统通常不允许长期关中断。

对于非实时操作系统来说，如果收到一个外部中断，那么操作系统在处理中断的整个过程中可能会一直关中断。但实时操作系统的通常做法是把中断作为一个事件通告给另外一个任务，interrupt handler 在处理完关键数据以后，立即打开中断，驱动的中断处理程序以一个高优先级任务的方式继续执行。

4. 系统级的服务也要保证实时性

对于一些系统级的服务，比如文件系统操作：

- 非实时系统会缓存用户请求，并不直接把数据写入设备，或者建立一系列的线程池，分发文件系统请求。
- 实时系统中允许高优先级的任务优先写入数据，在文件系统提供服务的整个过程中，高优先级的请求被优先处理，这种高优先级策略直到操作完成。

这种设计实际上会牺牲性能，但实时系统强调的是整个系统层面的实时性，而不是某一个点（比如内核）的实时性，所以系统服务也要实时。由于应用场景的差异，会出现有些用户需要实时性的驱动，有些用户需要高性能的驱动，因此实时操作系统实际上要提供多种形式的配置以满足不同实时性需求的用户。

5. 避免提供实时性不确定的 API

多数实时操作系统都不支持虚拟内存（page file/swap area）^[14]，主要原因是缺

页中断 (page fault) 会导致任务调度的不确定性增加。实时操作系统很多都支持分页, 但很少会使用虚拟内存, 因为一次缺页中断的开销十分巨大 (通常都是毫秒级), 波及的代码很多, 导致用户程序执行的不确定性增加。实时操作系统的确定性是一个很重要的指标, 在某些极端场景下, 甚至会禁用动态内存分配 (malloc/free), 来保证系统不受到动态的任务变化的干扰。

6. 提供针对实时系统调度的专用 API

比如 ARINC 653 标准中就针对任务调度等作出了一系列的规定, 同时定义了特定的 API 接口和 API 行为, 这些 API 不同于 POSIX API, 如果实时系统要在航空设备上使用, 就可能需要满足 ARINC 653 的规范。

7. 降低系统抖动

由于关中断等原因, 通常情况下, 操作系统的调度器不会太精确的产生周期性的调度, 比如 x86 早期的默认 60 的时钟周期 (clock rate), 抖动范围可能在 15-17ms 之间。但一个设计优秀的实时操作系统能把调度器的抖动降低到微秒甚至百纳秒一级, 在像 x86 这种天生抖动就很大的架构上, 降低系统抖动尤其重要。

8. 针对实时性设计的 SMP 和虚拟化技术

SMP (多核) 场景的实时调度是很困难的, 这里还涉及到任务核间迁移的开销。针对 SMP 场景, 多数实时操作系统的设计都不算十分优秀, 但相比普通操作系统来说, 实时操作系统的实时性通常更加优异。

同时实时操作系统的虚拟化能从 hypervisor^[15]层面上提供虚拟机级别的实时调度, 虚拟机上可以是另外一个实时系统, 也可以是一个非实时系统。

2.2 RISC-V 指令集

RISC-V 架构中包含两个指令集: 非特权指令集和特权指令集。这两个指令集分别用于不同的处理器模式和特权级别。

(1) 非特权指令集包含常规用途指令 (如 add, load 等), 用于运行普通应用程序, 例如图形界面应用、媒体应用等。这些指令没有访问处理器的特权资源和状态的权限, 因此被称为用户级指令集。在非特权模式下, 处理器只能执行非特权指令集中的指令。

(2) 特权指令集包含了一些需要特权级别才能执行的指令 (如 csrr, csrrw 等), 用于操作系统内核和其他需要访问特权资源和状态的特权软件。这些指令可以访

问处理器的特权资源和状态，如控制处理器的中断、异常、虚拟内存管理等。在特权模式下，处理器可以执行特权指令集中的指令。

2.2.1 RISC-V 非特权指令集

首先，对于 RISC-V 指令集来说，其采用如下的命名格式：

RV[###][abc...xyz]

表 2.1 RISC-V 指令集命名格式解析

标识符	说明
RV	用于标识 RISC-V 体系架构的前缀
[###]	{32,64,128}用于标识处理器的字宽
[abc...xyz]	标识该处理器支持的指令集模块集合

例如，RV32IMA 表达的是 RISC-V 架构 32 位字宽的指令集，支持 IMA 三个模块。同样的，RV64GC 指的是 RISC-V64 架构 64 位字宽的指令集，支持 GA 模块。

RISC-V 采用模块 ISA 的设计，即由 1 个基本整数（Integer）指令集+多个可选的扩展指令集组成。基础指令集 RV32I 是固定的，永远不会改变。其相关的一些指令集如下：

表 2.2 基本整数（Integer）指令集

基本指令集	描述
RV32I	32 位整数指令集
RV32E	RV32I 的子集，用于小型的嵌入式场景
RV64I	64 位整数指令集，兼容 RV32I。
RV128I	128 位整数指令集，兼容 RV64I 和 RV128I

表 2.3 扩展模块指令集

扩展指令集	描述
M	整数乘法（Multiplication）与除法指令集
A	存储器原子（Atomic）指令集
F	单精度（32bits）浮点指令集
D	双精度（64bits）浮点指令集，同时兼容 F
C	压缩指令集

.....	其它标准化和未标准化的指令集
-------	----------------

注：特定组合"IMAFD"被称为“通用（General）”组合，用英文字母 G 表示。

RISC-V 的非特权指令集规范定义了 32 个通用寄存器以及一个 PC 寄存器，通常 RISC-V 的 PC 寄存器是无法被访问的，是被屏蔽的，但是可以用特殊的方式去读。这些通用寄存器（General Purpose Registers）对 RV32I/RV64I/RV128I 都是一样的。如果实现支持 F/D 扩展则需要额外支持 32 个浮点（Float Point）寄存器。注意 RV32E 将 32 个通用寄存器缩减为 16 个。寄存器的宽度由 ISA 指定 RV32 的寄存器宽度为 32 位，依次类推。

每个寄存器具体编程是有特定的用途以及各自的别名。这个由 RISC-V Application Binary Interface（ABI）定义。下面是通用寄存器组的功能表：

表 2.4 通用寄存器组功能

寄存器名称	汇编名称	功能描述	调用返回后其值是否会保持不变
x0	zero	零寄存器	未定义
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	未定义
x4	tp	线程指针	未定义
x5	t0	临时寄存器，或者用作替代链接寄存器（见后续章节详述）	否
x6	t1	临时寄存器	否
x7	t2	临时寄存器	否
x8	s0/fp	该寄存器需要被调函数予以保存，或也可用作调用栈的栈指针	是
x9	s1	该寄存器需要被调函数予以保存	是
x10~x11	a0~a1	函数参数或返回值	否
x12~x17	a2~a7	函数参数	否
x18~x27	s2~s11	该寄存器需要被调函数予以保存	是
x28~x31	t3~t6	临时寄存器	否

RISC-V 是一种基于精简指令集(RISC)架构的指令集体系结构，其指令具有以

下特点：

1. 简洁明了：指令集精简，易于理解和实现。
2. 可扩展性好：支持可选的标准扩展或自定义扩展，可以满足不同应用场景下的需求。
3. 开放性高：完全开放的指令集体系结构，任何人都可以使用、实现和修改它。
4. 兼容性强：向后兼容性好，可以在不改变指令集的前提下进行升级和扩展。
5. 支持向量操作：内置向量指令，支持 SIMD（单指令流多数据流）操作，能够加速图像处理、科学计算等应用。
6. 高效节能：设计优化，执行效率高，能够降低功耗和热量产生。
7. 可移植性好：与不同的硬件平台和操作系统兼容性良好，可以在不同的设备上运行同一份代码。

根据其指令的特点，通常将非特权指令集的指令分成以下 6 种：

表 2.5 RISC-V 指令类型

指令类型	特点
R-type (Register)	每条指令中有三个 fields，用于指定 3 个寄存器参数。
I-type (Immediate)	每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12bits）。
S-type (Store)	每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12bits，但 fields 的组织方式不同于 I-type）。
B-type (Branch)	每条指令除了带有两个寄存器参数外，还带有一个立即数参数（宽度为 12bits，但取值为 2 的倍数）。
U-type (Upper)	每条指令含有一个寄存器参数再加上一个立即数参数（宽度为 20bits，用于表示一个立即数的高 20 位）。
J-type (Jump)	每条指令含有一个寄存器参数再加上一个立即数参数（宽度为 20bits）。

每种指令集格式 RISC-V 指令都有对应的编码格式，如下图：

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

图 2.1 RISC-V 指令编码格式

首先，32 个 bit 划分成不同的“域 (field)”，由 funct3/funct7 和 opcode 一起决定最终的指令类型，指令长度在 RV32I 中为 32bits，对齐长度也是 32bits。同时，在 RISC-V 中指令在内存中按照小端序排列的。另外要注意的是，funct3、funct7 后面的数字表示占了几个 bit，opcode 后两位一定是 11（也可以说是低两位），rd 表示目标寄存器，而 rs 表示源寄存器。

2.2.2 RISC-V 特权指令集

首先是特权级别 (Privileged Level)。RISC-V 的 Privileged Specification 定义了三个特权级别 (privilege level)。其中 Machine 级别是最高的级别，所有的实现都需要支持，刚上电的 CPU 运行在 Machine 中，此时不开放虚拟地址。其它特权级别如下图：

表 2.6 RISC-V 特权级别

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

与特权级别密切相关的是状态控制寄存器 Control and Status Registers (CSR)。在 CPU 的角度来说，他在不同的级别下各有一套独立的寄存器，这些不同的寄存器在不同的模式下是不能互相访问的，需要的是，第一点，不同的特权级别下分别对应各自的一套 Registers (CSR)，用于控制和获取相应 Level 下的处理器工作状态。第二点，高级别的特权级别下可以访问低级别的 CSR，反之则不可以。第三点，RISC-V 定义了专门用于操作 CSR 的指令。最后一点，RISC-V 定义了特定的指令可以用于在不同特权级别之间切换 (ECALL/EBREAK 命令)，相关的操作指令如图 2.2。

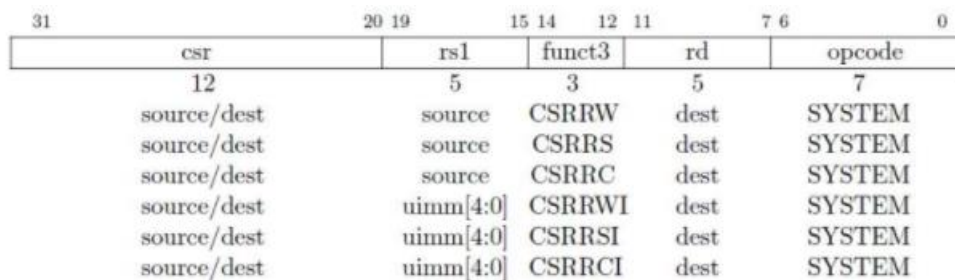


图 2.2 CSR 寄存器操作指令

常用的指令有：

(1) CSRRW (Atomic Read/Write CSR): CSRRW 先读出 CSR 中的值，将其按 XLEN 位的宽度进行“零扩展 (zero-extend)”然后写入 RD；然后将 RS1 中的值写入 CSR。

(2) CSRRS (Atomic Read and Set Bits in CSR): CSRRS 先读出 CSR 中的值，将其按 XLEN 位的宽度进行“零扩展 (zero-extend)”后写入 RD；然后逐个检查 RS1 中的值，如果某一位为 1 则对 CSR 的对应位置 1，否则保持不变。

常用的 CSR 寄存器如下：

(1) mtvec (Machine Trap Vector)

保存陷阱向量的设置。在官方实现中，其低 2 位为 MODE 域，高 32 位为 BASE 域。MODE 为 0，异常处理使用直接模式，所有异常会把 pc 设为 BASE；MODE 为 1，使用向量模式，所有同步异常会把 pc 设为 BASE，所有异步异常会使 pc 设为 BASE+4×cause；MODE 的其他值为 RSCV-V 保留。

(2) mstatus (Machine State)

跟踪并控制当前硬件线程(hart)的操作状态。其中第 3 位 MIE 机器模式全局中断使能位，置 1 时使能全局中断，置 0 时关闭全局中断，主要用于通过屏蔽中断来保证当前特权模式下有关中断处理程序的原子性。第 7 位为 MPIE 位，记录进入陷阱之前的中断使能位。11:12 位为 MPP 域保存进入异常前的特权模式。

(3) mcause (Machine Exception Cause)

用于异常发生时保存调入陷阱的原因。最高位为 1 时，由中断引起；为 0 时，由同步异常引起。它的低 31 位为异常代码，其中机器模式软件中断的代码为 3，机器模式计时器中断的代码为 7，机器模式外部中断的代码为 11。

(4) mscratch (Machine Scratch)

通常用于保存指向机器模式的硬件线程的局部上下文空间的指针，并在进入

机器模式陷阱处理程序前与用户寄存器交换。

(5) mepc (Machine Exception PC)

指向指向掉入陷阱时正在执行的指令。

(6) mie (Machine Interrupt Enable)

用于机器模式中断的使能。某位置 1 时,使能对应中断;置 0 时,屏蔽中断。

其中第 3 位 MSIE 为机器模式软件中断使能位,第 7 位 MTIE 为机器模式计时器中断使能位,第 11 位 MEIE 为机器模式外部中断使能位。

(7) mip (Machine Interrupt Pending) 用于保存目前正等待处理的中断,可用于查询中断的等待状态。其中各位功 能与 mie 中的各位对应,分别为 MSIP, MTIS, MEIP。

(8) mhartid 用于保存硬件 hart 的 id。

2.2.3 RISC-V Trap 处理流程

RISC-V 将异常与中断统称为 Trap, 当 Trap 刺激产生时, 会出现以下变化:

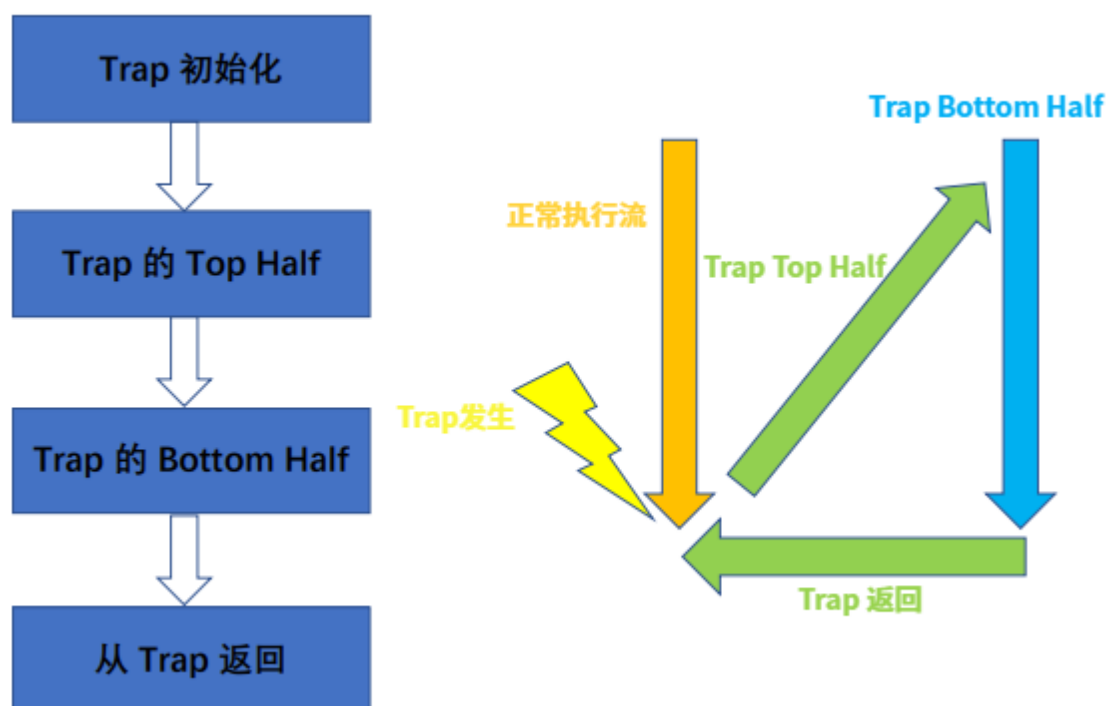


图 2.3 RISC-V 异常处理流程

Top Half 指的是 Trap 的上半部,通常由硬件处理,不受操作系统控制。Bottom Half 指的是 Trap 的下半部,有我们定义的执行逻辑。

在这个过程中，发生了以下的过程：

（1）Trap 初始化

设置入口函数的起始地址：通过写 `mtvec` 寄存器来实现。

（2）Trap 的 Top Half

Trap 发生，Hart 自动执行了如下状态切换：

首先把 `mstatus` 的 MIE 值复制到 `MPIE` 中，清除 `mstatus` 中的 MIE 标志位，效果是中断被禁止。之后设置 `mepc`，同时 PC 被设置为 `mtvec`。（需要注意的是，对于 `exception`，`mepc` 指向导致异常的指令；对于 `interrupt`，它执行被中断的指令的下一条指令的位置）。

- 发生异常时，本质上是给系统一个机会来在异常处理函数中将这个错误处理掉，当错误处理完毕后回去重新执行这条指令，就可能执行成功。
- 发生中断时，是执行当前指令时，外设请求中断发生，此时 CPU 会执行完当前的指令，然后切换执行中断，此时返回下一条指令，从程序执行顺序上讲仿佛什么都没发生。

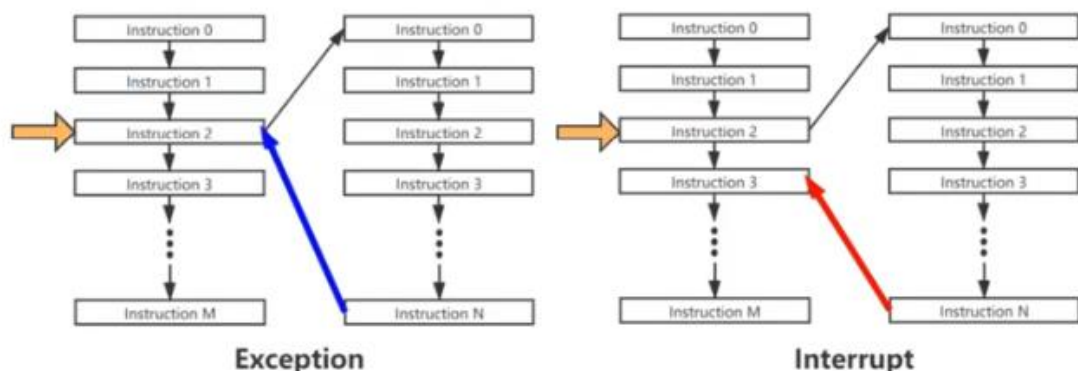


图 2.4 RISC-V Trap 处理流程

然后根据 `trap` 的种类设置 `mcause`，并根据需要为 `mtval` 设置附加信息。

最后将 `trap` 发生之前的权限模式保存在 `mstatus` 的 `MPP` 域中，再把 `hart` 权限模式更改为 `M`（也就是说无论在任何 `Level` 下触发 `trap`，`hart` 首先切换到 `Machine` 模式）。

（3）Trap 的 Bottom Half

首先要保存（`save`）当前控制流的上下文信息（利用 `mscratch`）。然后调用 C 语言的 `trap handler`。从 `trap handler` 函数返回，`mepc` 的值有可能需要调整，这个在 `trap handler` 函数中操作。之后恢复（`restore`）上下文的信息。最后执行 `MRET` 指令返回

到 trap 之前的状态

(4) 退出 Trap

退出 Trap 即编程调用 MRET 指令的过程，针对不同权限级别下如何退出 trap 有各自的返回指令 xRET ($x = M/S/U$)。以在 M 模式下执行 mret 指令为例，mret 会首先使当前 Hart 的权限级别 = mstatus.MPP；然后让 mstatus.MPP = U（如果 hart 不支持 U 则为 M）（恢复权限级别）。之后 mstatus.MIE = mstatus.MPIE；mstatus.MPIE = 1（恢复中断状态）最后，使 pc = mepc（跳转回原函数）。

2.3 K210 芯片简介

K210 包含 RISC-V 64 位双核 CPU，每个核心内置独立 FPU。K210 的核心功能是机器视觉与听觉，其包含用于计算卷积人工神经网络的 KPU 与用于处理麦克风阵列输入的 APU。同时 K210 具备快速傅里叶变换加速器，可以进行高性能复数 FFT 计算。因此对于大多数机器学习算法，K210 具备高性能处理能力。K210 内嵌 AES 与 SHA256 算法加速器，为用户提供基本安全功能。K210 拥有高性能、低功耗的 SRAM，以及功能强大的 DMA，在数据吞吐能力方面性能优异。K210 具备丰富的外设单元，分别是：DVP、JTAG、OTP、FPIOA、GPIO、UART、SPI、RTC、I²S、I²C、WDT、Timer 与 PWM，可满足海量应用场景。

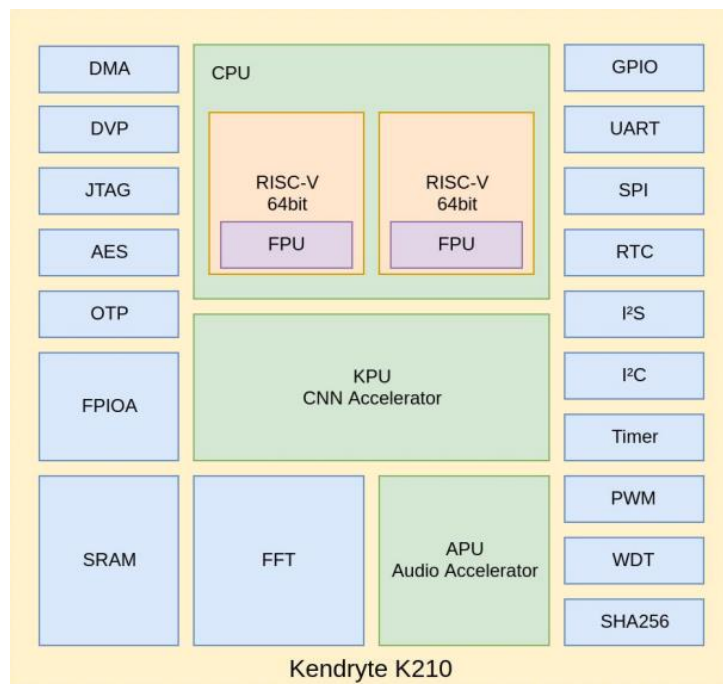


图 2.5 K210 功能框图

第三章 RVOS 的设计与移植方案

本章主要介绍 RVOS 的各部分设计方案以及 K210 移植过程中遇到的相关问题及解决方法。RVOS 由中科院开源 RISC-V 架 32 位实时操作系统改编而来，在其基础上进行了 64 位扩展与 K210 开发板移植，同时，对其各部分的功能也做了不同程序的增添与修改。RVOS 由内存管理，任务调度，串口（uart）输出，Trap 四大部分构成，其总体设计框图如下：

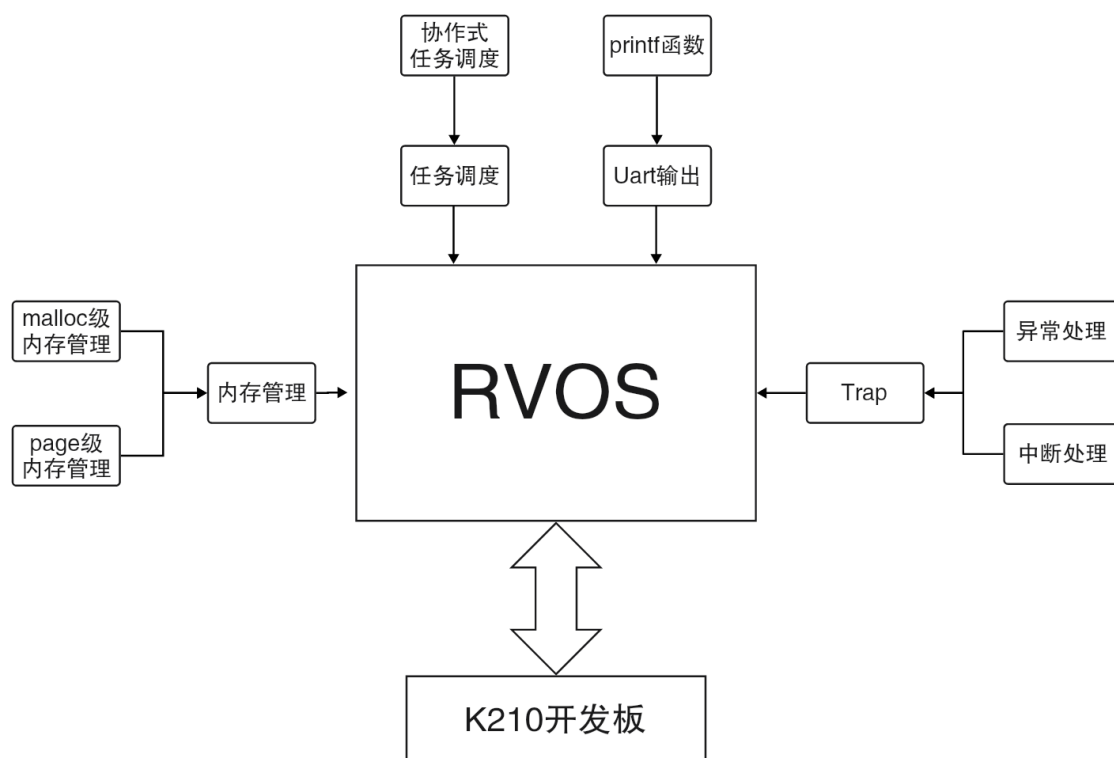


图 3.1 RVOS 设计框图

3.1 RVOS 初始化与 UART 输出

RVOS 在移植到 K210 之前是运行在 QEMU 上，所以首先要配置 QEMU，QEMU 的相关模拟参数如下：

表 3.1 QEMU 模拟参数

参数类别	参数内容
QEMU	qemu-system-riscv32
QFLAGS	-nographic -smp 1 -machine virt -bios none

之后要编写链接器脚本 `os.ld` 使链接器能正确链接，链接器脚本主要确定目标机架构与系统进入点（`_start`）。同时标识 `kernel` 地址以及可用空间。其次对 ELF 文

件各段进行划分。最后引出相关的段地址，供后续使用。

其次是编写系统进入函数（`_start`），该函数首先会休眠相关 hart。由于当前系统仅支持单核，所以要将其它 hart 休眠，仅运行 hart0。之后初始化.bss 段，将其全部清空为 0。然后设置内核栈，大小为 1024 字节，同时初始化堆栈指针。注意，当前系统的栈是放在代码段中。最后，切换的 C 代码的 `start_kernel` 函数，该函数为内核主函数。在 `start_kernel(void)` 函数中首先进行 uart 的初始化，使系统能对外输出。

对于 QEMU-virt 设备和 K210 开发板，其 uart 设备初始化完全不同，QEMU-virt 设备的 uart 型号为 td16550，根据其技术手册，`uart_init()` 函数中进行相关配置，目前仅配置了轮询输出方式，供 `uart_puts(char *s)` 函数输出系统相关信息。而 K210 使用 `uarths` 设备输出，要配置时钟频率进行初始化。通过 uart，RVOS 提供了 `printf` 函数来输出相关调试信息，其实现借用了 `mini-riscv-os` 的实现方法。

3.2 内存管理模块

内存管理上提供了两种内存分配方式。

1. Page 级分配：

Page 级的内存分配采用数组方式进行管理，结构如下：

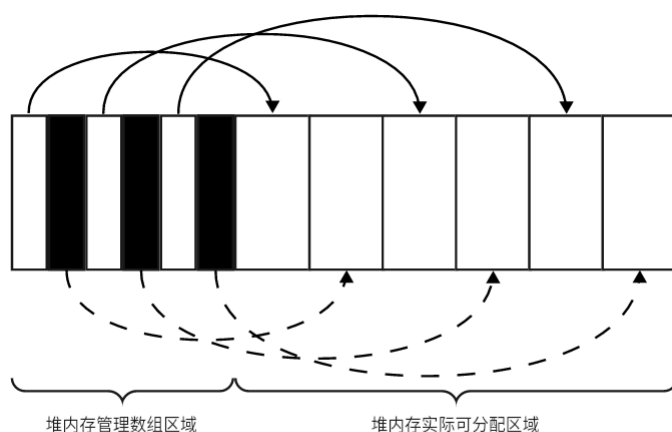


图 3.2 Page 级内存管理结构图

图中前方黑白相间的部分为内存管理的数组，后边白色部分代码内存块，黑色的数组项表示对应内存块此时被占用，白色表示对应内存块此时空闲。

内存管理的数组直接保存在堆底，大小为 8 页（ 8×4096 字节），每一个数组项管理一页（4096 字节）。这足够来管理 128MB（ $8 \times 4096 \times 4096$ ）的 heap 空间。

使用时，首先要调用 `page_init()` 函数进行初始化，该函数首先会计算可分配的内存页总数并保存到 `_num_pages` 中。然后刷新所有的数组项。最后对真正分配的堆页进行内存对齐，加快内存访问速度。

之后可以调用 `void *page_alloc(int npages)` 进行页分配，该函数分配时会依次检索数组，当遇到空闲页时，会检查该空闲页大小是否大于或等于 `npages`，如果符合，则将内的所有页刷新为占用，将最后一页进行标记，同时返回给用户首地址。

在释放时，调用 `void page_free(void *p)` 函数，根据其内存地址计算出首数组项位置，然后将数组项依次刷新为空闲，直到遇到最后一页为止（最后一页会有专门的标记）。

2.malloc 级的内存管理（参考 FreeRTOS 的 heap_4 实现方法）

malloc 级的内存分配采用双向链表方式进行管理，结构如下：

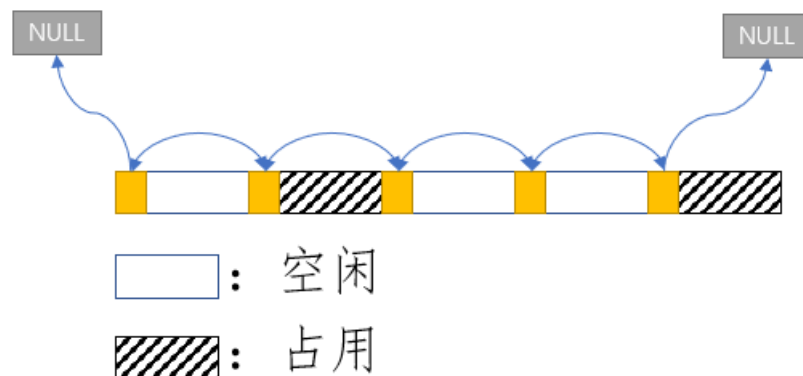


图 3.3 malloc 级内存管理结构图

每一个链表头直接保存在内存块的前 12 字节中。

使用时，首先调用 `void malloc_init()` 函数进行初始化，该函数会首先设置可分配内存大小，记录在 `_num_sizes` 中。然后初始化第一个空闲块，该块的大小是整个堆的可分配内存。

之后，可以调用 `void *malloc(size_t size)` 进行内存分配，分配时，函数会依次从第一个内存块开始向后检索，直到遇到第一个空闲的且大小足够的内存块为止，此时函数会将需要的 `size` 填入到链表头对应位置，同时设置 `flag` 标志位为占用。此后，如果该空闲块剩余的空间大于一个链表头大小，则将其创建为一个新的空闲块，加入到链表中。

在释放时，需要调用 `void free(void *ptr)` 函数，该函数会根据得到的空闲块实际可用空间的首地址指针，向前移动一个链表头大小的字节获得链表头。然后，该函数会检索该内存块左右相邻的内存块是否空闲，如果空闲，则将其合并（先检索后边的内存块再检索前边的内存块）。

3.3 协作式任务调度

任务调度的数据结构上使用优先级数组进行管理，同一优先级内的任务使用双向链表进行管理，最高支持优先级为 255。

下图是任务调度的整体结构：

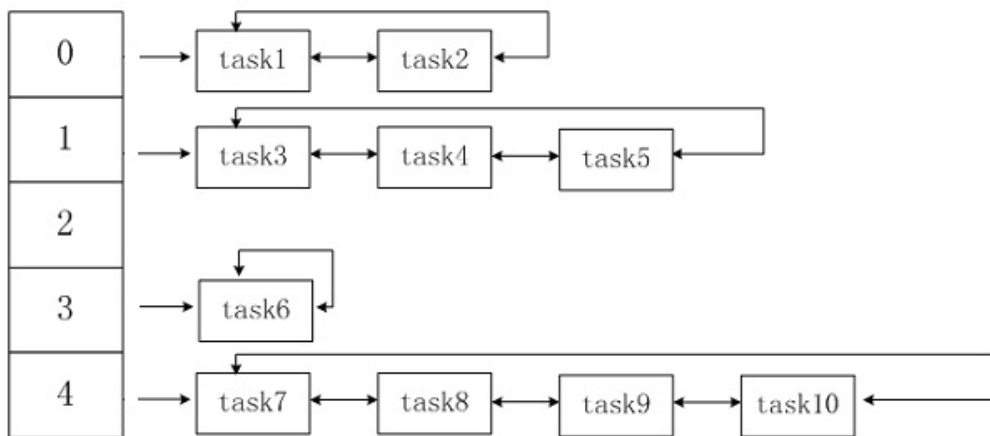


图 3.4 任务调度结构图

左边是优先级数组，其成员是一个指向任务结构体的指针，他指向相同优先级任务链表中的其中一个成员。

调度时，从最高优先级（0 级）依次搜索，遇到第一个非空的优先级链表时，比较原任务优先级与当前优先级，如果相同则进行链表内调度，否则调度最高优先级任务。

任务删除时，删除函数会判断，如果当前链表上只有一个任务，直接清空当前优先级，同时返回。而如果该任务是当前优先级的进入节点，则切换当前优先级进入节点。之后，删除函数内会回收当前优先级占有的任务空间，然后切入调度函数执行下一个任务。

需要注意的是，任务调度与任务删除享有自己独立的任务空间，调用时需先切换到其任务空间再进行操作。

为方便系统使用，这里提供了两个任务接口：

```
/* 提供任务调度接口 */
void task_yield(){
    switch_to(&schedule_context);
}
```

```
/* 提供任务退出接口 */
void task_exit(){
    switch_to(&exit_context);
}
```

函数内部会直接切换到任务调度与任务删除的任务空间，任务空间内通过 `now_task` 和 `now_priority` 接口进行调度与删除操作。

使用任务调度系统需要先调用 `void sched_init()` 函数进行初始化，函数中会进行会首先设置 `mscratch` 寄存器初值。然后初始化任务优先级数组。再设置 `schedule` 函数的上下文与 `exit` 函数的上下文。最后初始化 `now_priority` 值和 `now_task` 指针。

3.4 Trap 处理

RISC-V 的 Trap 处理函数主要存放在 `mtvec` (Machine Trap-Vector Base-Address) 寄存器中，其结构如下：

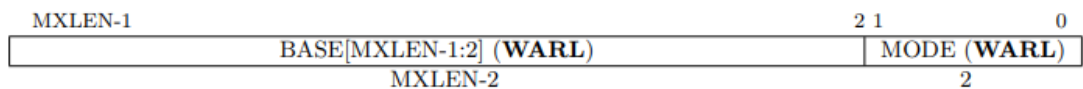


图 3.5 mtvec 寄存器结构

其中，`BASE` 为 `trap` 入口函数的基地址，必须保证四字节对齐（因为该寄存器最后两位被占用，所以他检索的地址会受限）。`MODE` 位用于设置进一步用于控制入口函数的地址配置方式，有以下两种方式：

表 3.2 Trap 处理模式

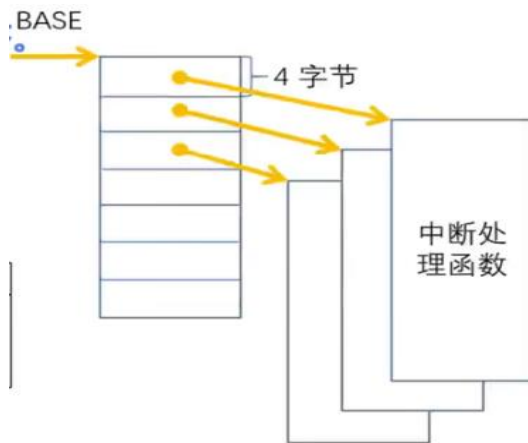
Trap 处理模式	说明
Direct	所有的 <code>exception</code> 和 <code>interrupt</code> 发生后，PC 都跳转到 <code>BASE</code> 指定的地址处
Vectored	Vectored: <code>exception</code> 处理方式同 Direct；但 <code>interrupt</code> 的入口地址以数组方式排列

可以这么理解：



MODE = Direct

图 3.6 Trap 的 Direct 模式



MODE = Vectored

图 3.7 Trap 的 Vectored 模式

在 Direct 方式中，我们把中断处理函数放在以 BASE 为起始的一块内存区域里面，在函数内进行分支判断对对应的情况进行执行。

而在 Vectored 方式中，我们将 BASE 指向一个数组，数组的内容是指向中断处理函数的指针，当发生中断时，CPU 根据相应的索引值查找对应的数组下标，找到对应的中断处理函数，Vectored 方式的处理效率肯定更高，更建议使用。

3.4.1 异常处理

异常处理中首先需要保存 (save) 当前控制流的上下文信息 (利用 mscratch 寄存器)。之后调用 C 语言的 trap handler 对异常进行处理，在 trap handler 函数中对当前异常状况进行判断并切换到对应的处理函数。从 trap handler 函数返回后，mepc 的值有可能有调整，要将其恢复。然后恢复 (restore) 上下文的信息。最后执行 MRET 指令返回到 trap 之前的状态。

3.4.2 中断处理

RISC-V 共有以下几种中断：

表 3.3 RISC-V 中断类型

中断类型	细分类别
本地 (Local) 中断	software interrupt
	timer interrupt
全局中断 (Global) 中断	external interrupt

可以认为是有软件中断、定时器中断、外部中断三种类型的中断。

中断发生时 Hart 自动执行如下状态转换：

1. 把 `mstatus` 的 MIE 值复制到 `MPie` 中，清除 `mstatus` 中的 MIE 标志位，效果是中断被禁止。
2. 当前的 PC 的下一条指令地址被复制到 `mepc` 中，同时 PC 被设置为 `mtvec`。注意当设置 `mtvec.MODE = vecored` 时 $PC = mtvec.BASE + 4 \times \text{exception-code}$ 。
3. 根据 `interrupt` 的种类设置 `mcause`，并根据需要位 `mtval` 设置附加信息。
4. 将 `trap` 发生之前的权限模式保存在 `mstatus` 的 `MPP` 域中，再把 `hart` 权限模式更改为 M。

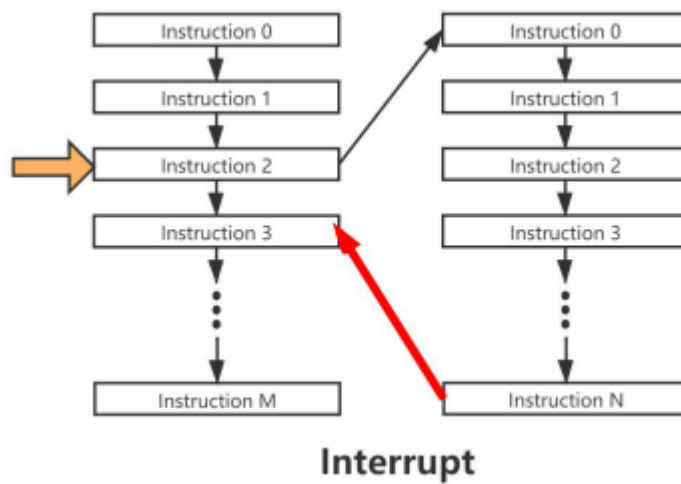


图 3.8 中断触发模式

在大部分 RISC-V 设备中外部中断都是由 PLIC（Platform-Level Interrupt Controller）控制，其控制结构如下：

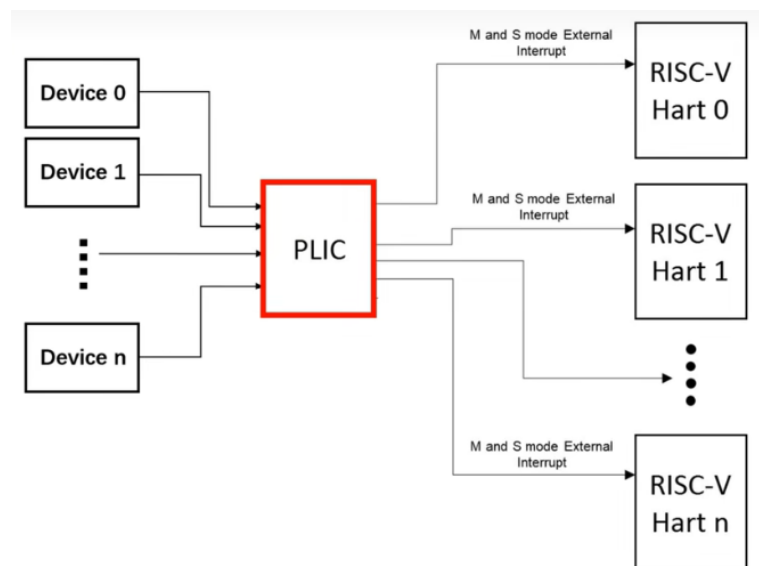


图 3.9 PLIC 控制结构

PLIC 是集成在 SOC 内的中断控制器。左边的 Device 都是中断源（Interrupt

Source)。当中断源产生多个中断时, PLIC 会通过类型、优先级等判断, 只允许一个中断进入。在编程时, 要初始化 PLIC, 确定外设的中断源, 同时根据相关寄存器编程规范设置优先级, 使能该中断标志位, 同时设置阈值。在中断开始与结束, 要读写 claim/complete 寄存器来确定中断号与完成该中断, 这些都在实现函数中有相应反应。

3.5 RVOS 的 64 位扩展

最开始开发的系统为 32 位, 为了移植到 K210 上, 需要对其进行 64 位扩展。在这个过程中, 需要注意的点如下:

- (1) 存储 32 位数据的变量修改为 64 位, 如存储通用寄存器数据相关的变量。
- (2) 汇编部分代码中, 将相应的加载指令 ld/sd 变为 lw/sw 以支持 64 位扩展。
- (3) 64 位需要更大的栈空间, 需要将相应的空间扩容。
- (4) 将编译选项中 -mcmdol=medlow 改为 -mcmdol=medany, 原因如下:

对 -mcmdol=medlow 来说从编译结果上看, -mcmdol=medlow 使用 LUI 指令取符号地址的高 20 位。LUI 配合其它包含低 12 位立即数的指令后, 可以访问的地址空间是 -2GiB ~ 2GiB。对于 RV32, 就是 0x00000000 ~ 0xFFFFFFFF, 就是说可以访问任意地址了。然而对于 RV64 而言, 能访问的就是 0x0000000000000000 ~ 0x000000007FFFFFFF, 以及 0xFFFFFFFF800000000 ~ 0xFFFFFFFFFFFFFFFF 这两个区域, 前一个区域即 +2GiB 的地址空间, 后一个区域即 -2GiB 的地址空间。其它地址空间就访问不到了。

而对 -mcmdol=medany 来说从编译结果来看, -mcmdol=medlow 使用 AUIPC 指令取符号地址的高 20 位。AUIPC 配合其它包含低 12 位立即数的指令后, 可以访问当前 PC 的前后 2GiB (PC - 2GiB ~ PC + 2GiB) 的地址空间。对于 RV32, 能访问的还是 0x00000000 ~ 0xFFFFFFFF 这个区间, 也是访问任意地址。然而对于 RV64, 取决于当前 PC 值, 能访问到是 PC - 2GiB 到 PC + 2GiB 这个地址空间。假设当前 PC 是 0x1000000000000000, 那么能访问的地址范围是 0x0000000080000000 ~ 0x100000007FFFFFFF。假设当前 PC 是 0xA000000000000000, 那么能访问的地址范围是 0x9000000080000000 ~ 0xA00000007FFFFFFF。所以, 在 64 位下应该选择 -mcmdol=medany。

3.6 基于 K210 的 RVOS 移植

移植过程主要分为两步：

(1) 对 RVOS 进行定制性改造。

首先要注意 K210 使用高速 UART (UARTHS) 作为调试串口，与普通的 UART 配置不一样的点在于，UARTHS 需要使用获取系统时钟频率来进行设置。K210 的高速 UART 提供了多种分频比较灵活的时钟配置选项，如系统时钟分频器 (SYSCTL_CLK_DIV)，核心时钟分频器 (CLK0DIV)，分数时钟分频器 (PLL0) 等，具有较好的时钟管理能力。此外，还提供了 DMA 接口，可通过 DMA 引擎实现高效的数据传输和处理。

其次 K210 的 sp 寄存器强制数据 64 位对齐，所以，需要在代码中对相应的数据以及函数进行对其处理。

同时，对于 QEMU-Virt 设备与 K210 的 memory map 不同，应将对应部分进行修改。如下：

表 3.4 QEMU-Virt 与 K210 的 memory map

外设	地址
QEMU-virt uart	0x10000000UL
K210 uarths	0x38000000U
QEMU-virt plic	0x0c000000L
K210 plic	0x0C000000ULL

最后，K210 的 Trap 仅支持 Direct 模式，所以 K210 的中断必须进入统一 trap 处理函数后进行处理。

(2) 将编译产生的 os.bin 文件烧录到 K210 内存中。

在编译完成后，bin 文件夹会生成 os.elf 文件与 os.bin 文件，os.bin 为二进制文件，用于直接烧录到 K210 开发板内存中启动。烧录工具使用 K210 官方提供的 Kflash。选择波特率 115200，直接将 os.bin 文件烧录进内存中，之后 Kflash 将自动打开一个串口输出框，可以在其中查看串口调试信息并进行交互。

第四章 RVOS 各模块设计细节

该章节将以流程图的形式介绍 RVOS 各模块的重要部分的代码设计原理以及相关数据结构。

4.1 系统初始化

进入系统主函数 `start_kernel` 之前，首先要通过 `_start` 函数进行系统的初始化，`_start` 函数类型一个小型的 `bootload`，它主要会进行以下操作：

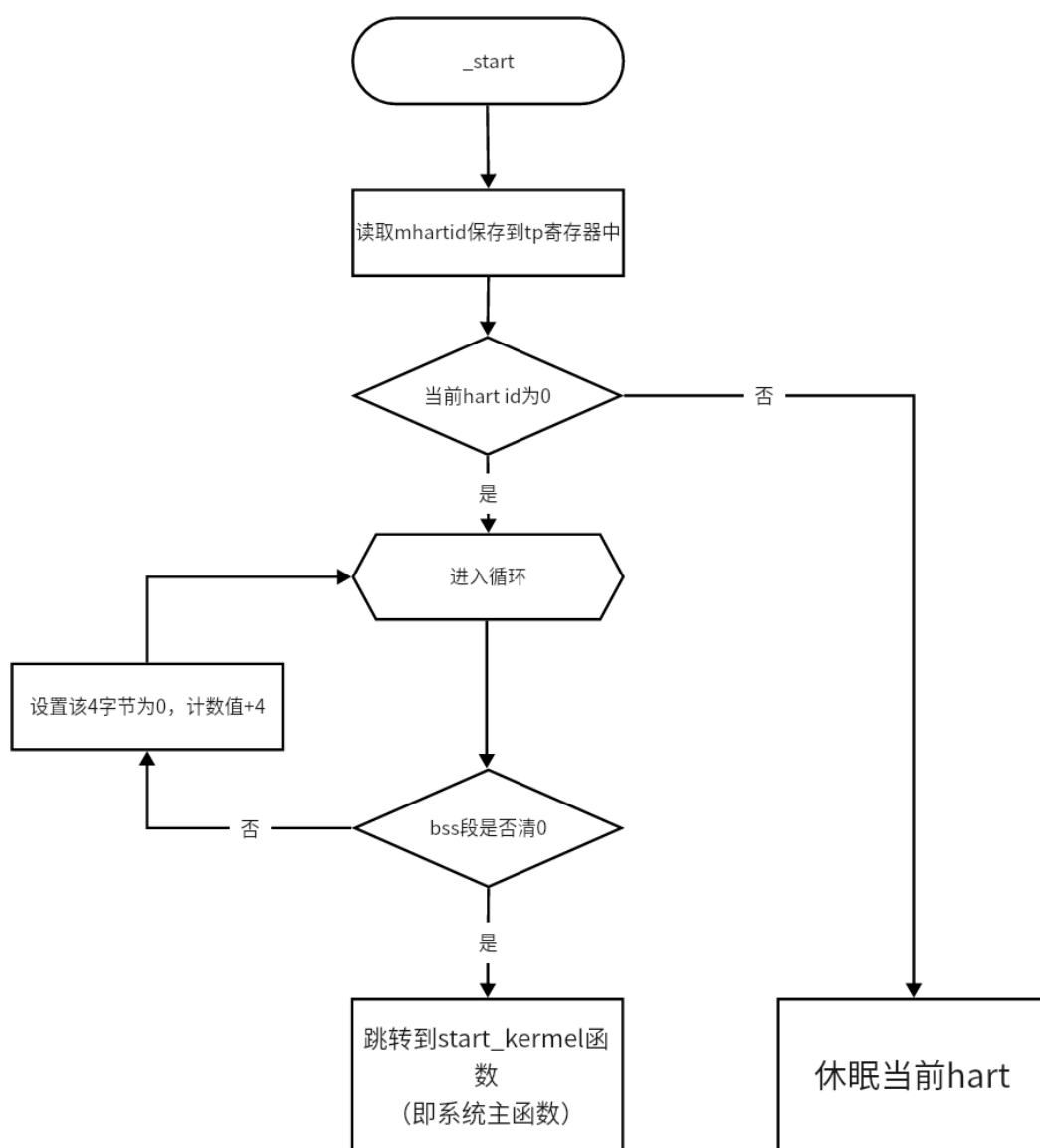


图 4.1 `_start` 函数流程图

首先要注意一点，对于一个 CPU 来说，在启动后，所有的 `hart` 都会运行内存中的程序，所以在进入 `_start` 函数后，由于当前系统仅支持单核，要将其它 `hart` 休

眠，仅运行 hart0。程序会首先读取 mhartid 的值保存到 tp 寄存器中，为后续的使用做准备。之后，检查当前 hart id 是否为 0，如果不为零则休眠当前 hart。

之后，进入循环，判断.bss 段是否为空。这一步是通过 a0, a1 两个寄存器实现的，运行前先将.bss 段底地址加载到 a0 中，将.bss 段顶地址加载到 a1 中，然后对 a0 的地址赋 0 后加 4，知道 a0 与 a1 相等。

赋值结束后，将内核栈的地址赋给 sp 指针，然后就可以跳转到 start_kernel 函数，该函数为内核主函数。

该部分的源码如下：

```

_start:
    #设置hart, 仅hart0 运行, 其它hart 休眠
    csrr t0,mhartid    #读取当前hart id
    mv tp,t0           #将hart id 保存在tp 寄存器中以后使用
    bnez t0,park        #如果当前hart id 不为0, 进入休眠
    #设置BSS section 的所有byte 为0
    la a0, _bss_start
    la a1, _bss_end
    bgeu a0,a1, _code_continue #无符号方式比较, a0>=a1 时跳转
_set_zero:
    sw zero, (a0)
    addi a0,a0,4
    bltu a0,a1, _set_zero #无符号方式比较, a0<a1 时跳转
_code_continue:
    #设置栈, 栈是从底部开始生长的, 所以我们将栈指针设置到栈底
    slli t0,t0,10      #左移hart id 10 位, 低10 位为1024 字节, 正好为每个
hart 的栈空间
    la sp, stacks + STACK_SIZE #设置栈指针指向栈底
    add sp,sp,t0        #将栈指针移动到当前hart 的栈底
    j start_kernel      #hart 0 跳转到c
park:
    wfi                #RISC-V 架构定义的一条休眠指令
    j park
stacks:
    .skip    STACK_SIZE #为所有hart 分配空间
    .end     #文件截止

```

4.2 内存管理模块

page 级内存管理使用数组来管理每页内存，数组项如下：

```

struct Page{
    uint8_t flages;
};

```

Page 具体结构可以简化如图 4.2 所示。每一个数组项中，只保留一个 8bit 的

flags 的标志位，这个标志位中，bit0 标识该页是否已分配，1 表示已分配，0 表示未分配；而 bit1 表示该页是否为内存块的最后一页，同样的，1 表示是最后一页，0 表示不是最后一页。



图 4.2 Page 结构

而 malloc 级内存管理是由链表控制，内存块信息直接保存在链表头中，其结构如下：

```

struct Block{
    reg_t *front;
    reg_t *next;
    reg_t size_flag;
};
  
```

同样，该结构体的结构可以简化为图 4.3。Block 首先有两个指向前后结构体的指针 front、next，其是组成 malloc 级内存管理的重要部分。之后，size_flag 标志位用来保存 Block 的相关信息。bit0 标识该块是否空闲，1 表示该块已被使用，0 表示该块空闲；剩余的标志位用来储存该块的大小。同时，Block 的实例的大小是确定的，其大小在 64 位的系统中固定为 24 字节。

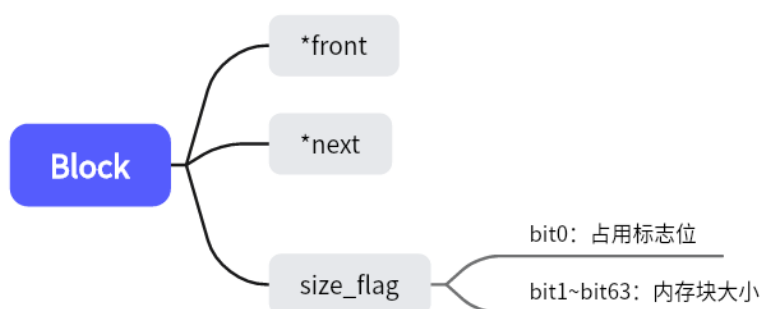


图 4.3 Block 结构

Block 之后，malloc 级内存管理最关键的部分便是 malloc 函数。以下流程图更加直观的描述 malloc 级内存管理的内存分配流程：

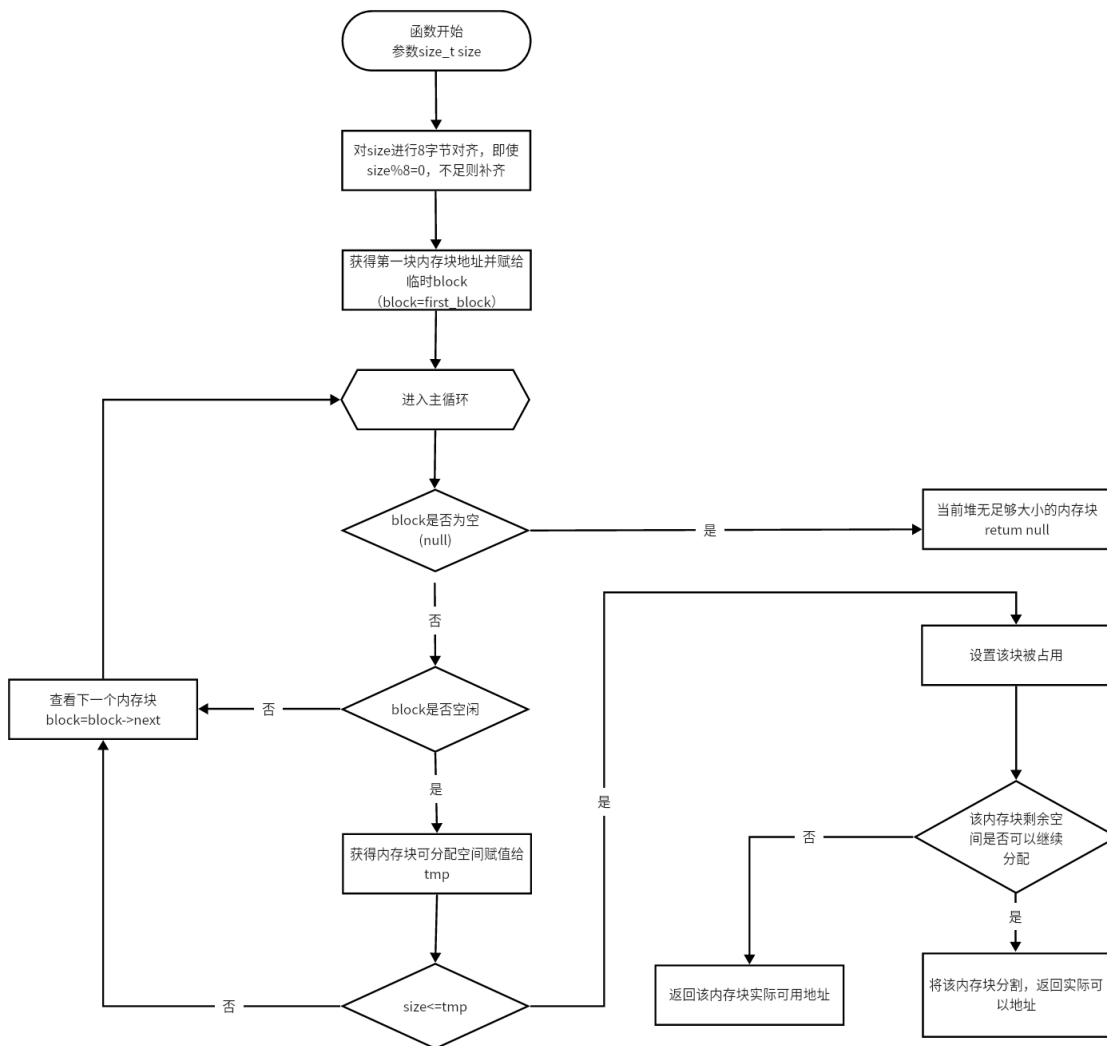


图 4.4 malloc 函数流程图

`malloc` 函数开始时, 首先对要分配的空间进行 8 字节对齐, 这是为了使内存块符合对齐原则, 防止 K210 由于对齐异常问题报错。之后, 通过全局变量 `first_block` 获取头结点的指针然后进入循环来获取空闲的内存块。注意, `block` 为空时表明内存中已经没有足够大小的内存块, 此时返回 `null`。获取到内存块后, 检查该内存块的大小是否大于需要分配的内存 `size`。如果小于, 则继续进行循环。当大于时, 将该内存块标志位设置为占用, 之后, 检查该内存块是否还有剩余空间, 如果有, 则将其分割, 之后, 返回分配到的地址。

`free` 函数的重点在于前后两块空闲内存块的合并, 其用流程图表示如下:

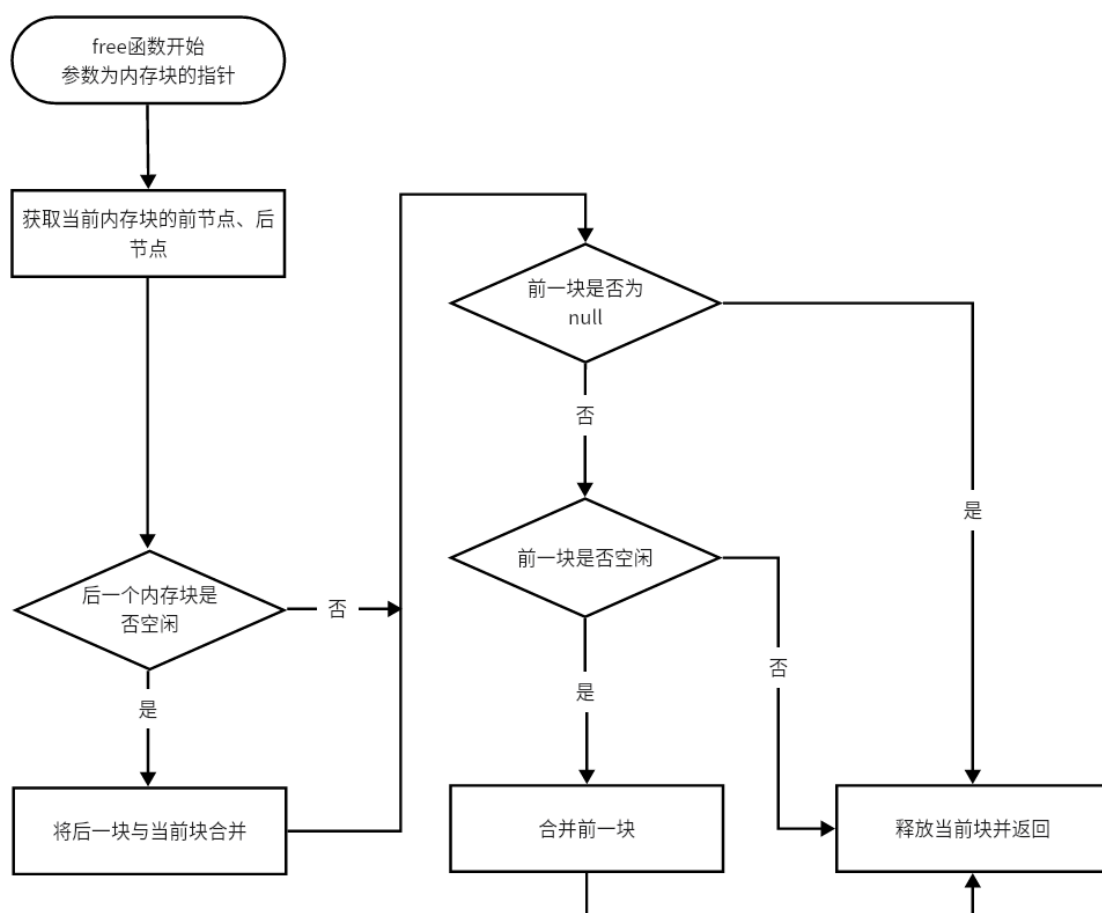


图 4.5 free 函数流程图

free 函数进入故意，首先会获取当前内存块的前后节点地址，为后续内存块合并做准备。之后，程序会先检查后一块内存块是否空闲，如果空闲，则将后一内存块与需要释放的内存块合并，合并即将当前内存块的后向指针越过后一节点指向下一节点。之后，程序会检查前一内存块是否为 null，这一步是防止内存异常访问到堆区以外的地址，当被释放内存块是堆区头结点的话，就会出现这种情况。如果前一内存块为 null，则直接释放当前内存块并返回。否则，进一步检查前一块内存释放空闲，如果空闲，进行合并，否则释放当前块并返回。前面程序完成后，最后都会释放当前块并返回。

4.3 任务调度器

为实现任务调度器，首先要对每个任务都有统一的数据结构来描述。结构体 Task 便是描述任务模型的任务结构体，其结构如下：

```

/* 定义每一个任务的结构体 */
struct Task
{
    uint8_t task_stack[STACK_SIZE];
    uint8_t priority;
    struct context ctx_tasks;
    struct Task *front;
    struct Task *next;
};

```

任务结构体的结构图如图 4.6 所示，由任务栈、任务优先级、任务上下文、前向任务指针、后向任务指针，五部分组成。

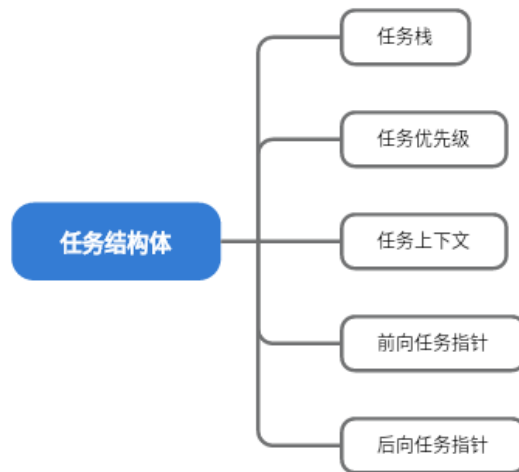


图 4.6 任务结构体结构图

任务栈是每个任务所使用的独立栈段，其大小为 4K 字节。任务优先级用于标明当前任务的优先级，其在初始化注册任务时标明。任务上下文用于进行上下文切换时使用，其结构由上下文结构体 context 描述。

<pre> /* task management */ struct context { /* ignore x0 */ reg_t ra; reg_t sp; reg_t gp; reg_t tp; reg_t t0; reg_t t1; reg_t t2; reg_t s0; reg_t s1; reg_t a0; reg_t a1; </pre>	<pre> reg_t a6; reg_t a7; reg_t s2; reg_t s3; reg_t s4; reg_t s5; reg_t s6; reg_t s7; reg_t s8; reg_t s9; reg_t s10; reg_t s11; reg_t t3; reg_t t4; </pre>
---	--

<pre> reg_t a2; reg_t a3; reg_t a4; reg_t a5; </pre>	<pre> reg_t t5; reg_t t6; }; </pre>
--	-------------------------------------

当上下文切换时，任务上下文保存通用寄存器的值。前向任务指针和后向任务指针共同构成同一优先级中的优先级链表结构。

对于任务调度而言，最重要的部分在于调度器的实现，RVOS 的调度器结构图如下：

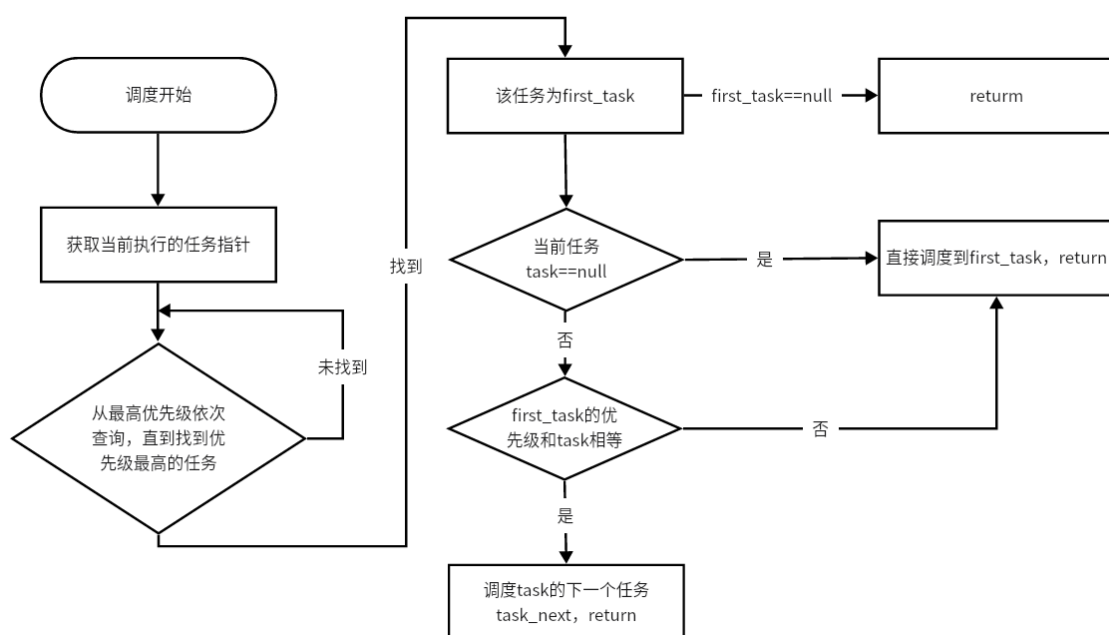
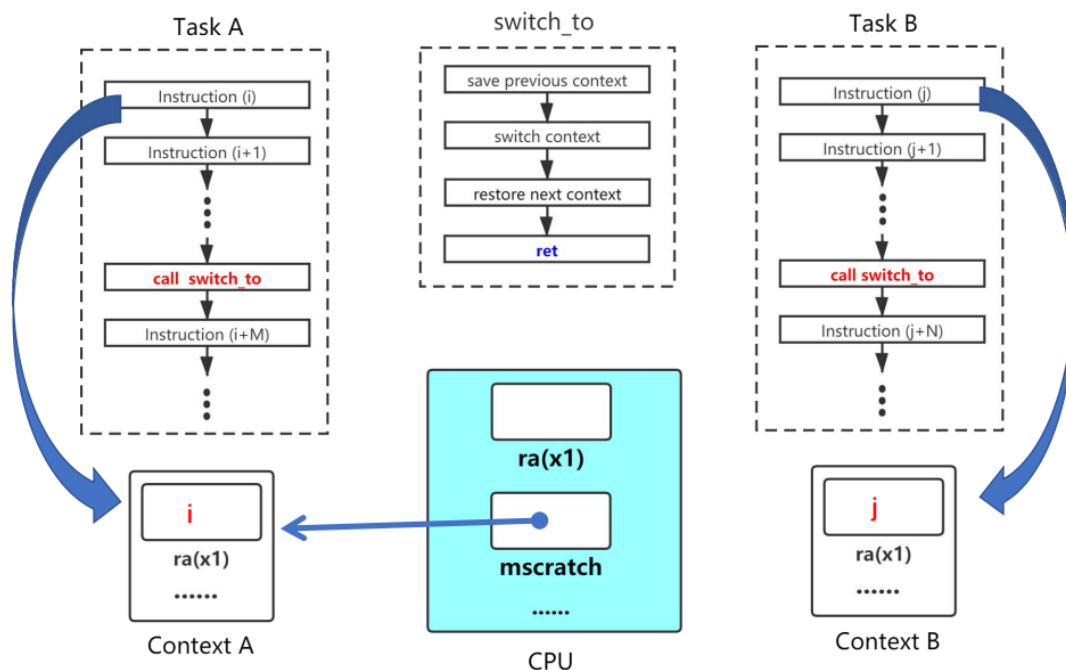


图 4.7 RVOS 任务调度器逻辑

调度时，程序从最高优先级（0 级）依次搜索，此时会出现两种情况：第一种，当最终遍历到优先级链表都没有任务时，此时指向任务的指针为 `null`，这表明此时系统中没有任务等待调度；第二种，如果找到，则该任务是最高优先级链表的第一个任务，将其赋给 `first_task`。之后，比较当前任务与 `first_task` 的优先级，如果相等，则证明当前是同一优先级链表内调度，直接调度当前优先级的后一节点上的任务；否则，直接调度到最高优先级任务 `first_task`，最后返回。

具体调度时，由汇编函数 `switch_to` 进行两个任务之间的切换。

`switch_to` 首先会保存上一个任务的上下文。之后切换任务。然后加载新任务的上下文，最后通过 `ret` 返回（在切换上下文时，`ra` 寄存器也进行了更改）。具体模型如图 4.8 所示。

图 4.8 `switch_to` 调度模型

Context A 和 Context B 分别表示存储在内存中的上下文 A 和上下文 B。

处理过程：首先 `ra` 分别填入两个任务的第一条指令的位置。此时，`mscratch` 指向任务 A 的上下文。当发生 `call` 时，`ra` 首先会保存任务返回值。进入 `switch_to` 后，会保存寄存器值到上下文中。之后，`mscratch` 切换，指向任务 B 的上下文。最后，加载 B 的上下文，然后 `ret` 根据 `ra` 值跳转到任务 B 的第一条指令。其关键部分源码如下：

```
# void switch_to(struct context *next);
# a0: 指向下一个任务的上下文的指针
.globl switch_to
.align 3
switch_to:
    csrrw    t6,mscratch,t6    # 交换 t6 和 mscratch 的值
    beqz     t6,1f             # 注意：第一次调用 switch_to() 函数时，
    # mscratch 寄存器被初始化为 0（在 sched_init() 中），
    # 这使得 t6 为 0，此时直接跳转对 mscratch 进
    行初始化
    reg_save t6                 # 保存前一任务的上下文
    # 保存实际的 t6 寄存器
    mv       t5,t6              # t5 指向当前任务的上下文
    csrr     t6,mscratch         # 获取 t6 的实际值
    sd       t6, 240(t5)        # 将 t6 的值保存在正确的位置（我们定义的是上下文偏移 120）
1:
    # 设置 mscratch 寄存器的值指向新任务的上下文
    csrrw    mscratch, a0
```

```

# 加载所有的通用寄存器
# 此时我们需要使用 t6 存放上下文地址,
# 因为如果使用 a0 那么在加载过程中就会被覆盖掉
# 而 t6 是在最后加载
mv    t6, a0
reg_restore t6
ret

```

4.4 Trap 处理

Trap 的处理在 QEMU 与 K210 上有不同的方式, 由于 K210 不支持 Trap Vector 模式, 所以只能采用 Trap Direct 模式。两种模式的不同在第三章有叙述。但由于 K210 在中断实现上存在问题, 目前中断与异常仅在 QEMU 上进行了全套的实现。在 QEMU 上, 硬件会自动根据中断与异常进入相应的处理程序。中断与异常的汇编部分处理接近相同, 区别只在于最终进入不同的 c 语言处理函数。这里仅介绍异常的汇编代码函数 `trap_vector` 做介绍。

该部分使用汇编实现, 定义在 `entry.S` 中的 `trap_vector` 函数, 其关键部分代码如下:

```

# 机器模式下的异常出现在这里
.globl trap_vector
# trap 向量基址必须始终在 8 字节边界上对齐
.align 3
trap_vector:
    # 保存当前上下文
    csrrw    t6, mscratch, t6    # 交换 t6 和 mscratch 的值
    reg_save t6                  # 保存前一任务的上下文
    # 保存实际的 t6 寄存器
    mv    t5, t6                # t5 指向当前任务的上下文
    csrr    t6, mscratch        # 获取 t6 的实际值
    sd    t6, 240(t5)          # 将 t6 的值保存在正确的位置 (我们定义的是上下文偏移 120)
    # 将上下文指针恢复到 mscratch
    csrw    mscratch, t5
    # 调用在 trap.c 中的 trap 处理函数
    csrr    a0, mepc
    csrr    a1, mcause
    call    trap_handler
    # trap_handler 会通过 a0 返回返回地址
    csrw    mepc, a0
    # 恢复上下文
    csrr    t6, mscratch
    reg_restore t6

```

```
# 回到trap之前做的事情
mret
```

程序开始后，由于 `mscratch` 寄存器是特权寄存器，只能用特殊的读写指令来操作，所以程序首先会交换 `t6` 与 `mscratch` 的值，此时利用 `t6` 来保存当前任务的上下文，之后，将 `mscratch` 的值加载到 `t6` 中，保存实际的 `t6` 值。此时，实际上下文的指针保存在 `t5` 中，这时可以将 `t5` 重新加载到 `mscratch` 中。之后，将 `mepc`，`mcause` 的值加载到 `a0`，`a1` 中，作为 `trap` 处理函数 `trap_handler` 的参数，然后程序跳转到 `trap_handler` 函数执行。执行完毕返回后，恢复任务上下文，之后执行 `mret` 函数返回。具体流程如图 4.9 所示。

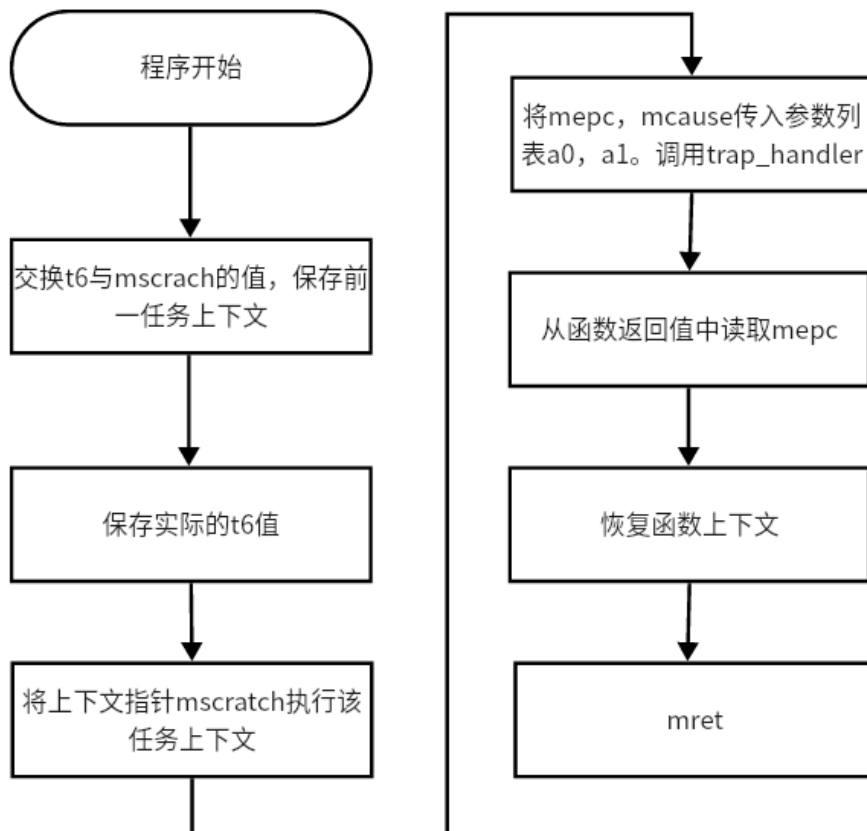


图 4.9 `trap_vector` 函数流程图

第五章 RVOS 功能模块测试

本章节将对 RVOS 各功能模块进行测试，并对展示各功能模块关键部分的相关代码。

5.1 测试环境搭建

在移植 K210 之前，首先使用 QEMU-virt 设备进行系统测试。virt 平台的全称是 QEMU RISC-V VirtIO Board，由 SiFive 公司设计，并且包含 16550a UART 和 VirtIO MMIO 作为外设和 I/O 接口。可以按照如下命令配置测试平台：

```
$ sudo apt update
$ sudo apt install build-essential gcc make perl dkms git gcc-riscv64-unknown-elf gdb-multiarch qemu-system-misc
```

进入 Linux 后，make platform=K210 或 QEMU 选择相应的平台，然后会在 bin 文件夹输出 os.bin 与 os.elf 系统文件。在 make run 命令中 QEMU-virt 设备将采用 os.elf 启动系统。相关代码如下：

```
run: $(OUT_PUT)/os.elf
    @${QEMU} -M ? | grep virt >/dev/null || exit
    @echo "Press Ctrl-A and then X to exit QEMU"
    @echo "-----"
    @${QEMU} ${QFLAGS} -kernel $(OUT_PUT)/os.elf
```

同时，QEMU-Virt 设备支持 GDB 调试，使用 make debug 命令可以进行调试环境，相关代码如下：

```
.PHONY : debug
debug: $(OUT_PUT)/os.elf
    @echo "Press Ctrl-C and then input 'quit' to exit GDB and QEMU"
    @echo "-----"
    @${QEMU} ${QFLAGS} -kernel $(OUT_PUT)/os.elf -s -S &
    @${GDB} $(OUT_PUT)/os.elf -q -x gdbinit
```

还可以使用 make code 来反汇编查看相关代码：

```
.PHONY : code
code: $(OUT_PUT)/os.elf
    @${OBJDUMP} -S $(OUT_PUT)/os.elf | less
```

进行 K210 移植后，使用 K-FLASH 进行代码烧录。注意，K210 使用 CH340 芯片作为 USB 转串口芯片，所以使用前需挂载 CH340 的驱动。烧录界面如下：

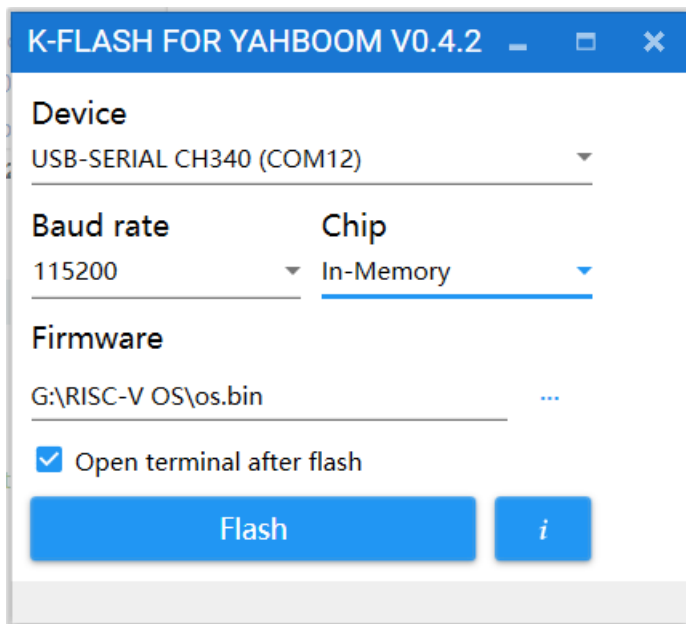


图 5.1 K-FLASH 界面

K210 烧录成功后，K-FLASH 将自动打开一个串口输出界面，如图 5.2 所示。所输出的信息都是 RVOS 通过串口输出的相关信息，其记录了内存各段起始位置，各功能模块测试信息等，是后文功能测试的基础。

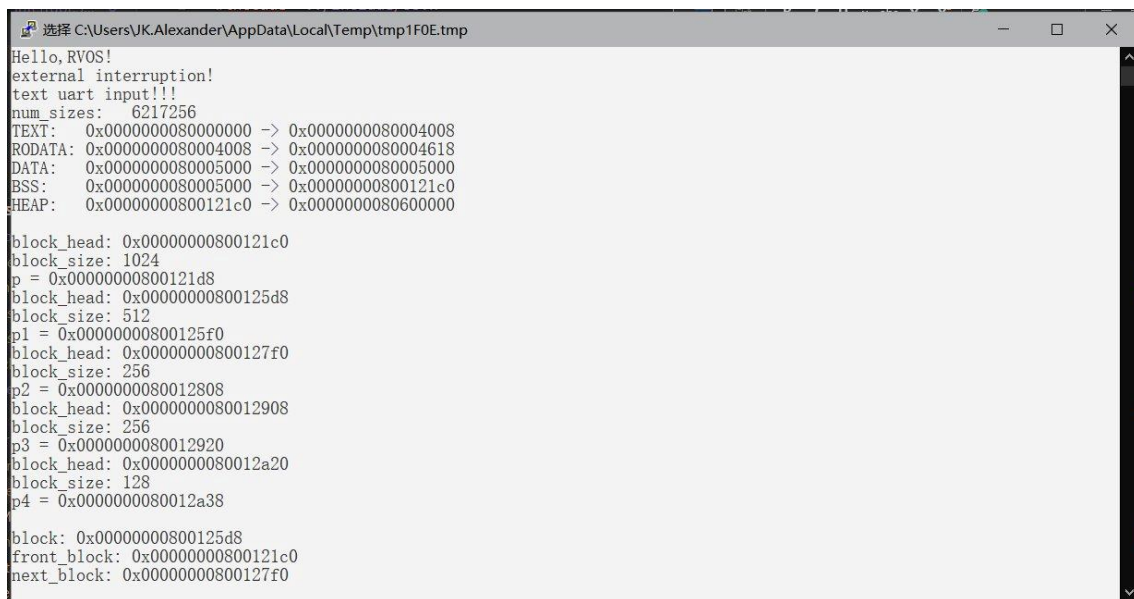


图 5.2 K210 串口输出界面

5.2 RVOS 初始化测试

系统启动后，首先会将_start 函数加载到内存起始位，这个位置通常在地址映射表中为 0x80000000，QEMU-virt 设备的内存映射如图 5.3 所示。

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] = { 0x0, 0x100 },
    [VIRT_MROM] = { 0x1000, 0xf000 },
    [VIRT_TEST] = { 0x100000, 0x1000 },
    [VIRT_RTC] = { 0x101000, 0x1000 },
    [VIRT_CLINT] = { 0x2000000, 0x10000 },
    [VIRT_PCIE_PIO] = { 0x3000000, 0x10000 },
    [VIRT_PLIC] = { 0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] = { 0x10000000, 0x100 },
    [VIRT_VIRTIO] = { 0x10001000, 0x1000 },
    [VIRT_FLASH] = { 0x20000000, 0x4000000 },
    [VIRT_PCIE_ECAM] = { 0x30000000, 0x10000000 },
    [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    [VIRT_DRAM] = { 0x80000000, 0x0 },
};
```

图 5.3 QEMU-Virt 设备内存映射图

在_start 完成基本初始化后, 将跳转到 start_kernel 后, 首先进行的便是 uart 的初始化, QEMU-Virt 设备与 K210 的初始化方式有所不同, 二者都在 uart_init()函数中进行初始化。对于 QEMU-Virt 设备, 初始化代码如下:

```
/* disable interrupts. */
uart_write_reg(IER, 0x00);
/*
 * 设置波特率。
 * 注意DLL 与DLM 和RHR,THR,IER 有相同的地址, 为了选择我们要写的寄存器,
 * 要设置LCR 寄存器
 * 第7 位为1
 * 波特率数值见表, 我们使用 38.4K 时的 1.8432 MHZ 的晶体, 所以对应的值是
 * 3。
 * 由于除数寄存器是两个字节 (16 位), 所以要分开设置, 低位DLL 设为3, 高位
 * DLM 设为0
 */
uint8_t lcr = uart_read_reg(LCR);
uart_write_reg(LCR, lcr | (1 << 7));
uart_write_reg(DLL, 0x03);
uart_write_reg(DLM, 0x00);
/*
 * Continue setting the asynchronous data communication format.
 * 设置异步数据通信格式。
 */
lcr = 0;
uart_write_reg(LCR, lcr | (3 << 0));
/*
 * enable receive interrupts. (使能接收中断)
```

```

*/
uint8_t ier = uart_read_reg(IER);
uart_write_reg(IER, ier | (1 << 0));

```

K210 初始化时，首先会通过 `k210_get_clk_freq()` 函数来获取当前 `uarths` 设备所需的频率，对其进行处理后填入到变量 `div` 中。之后，再对 `uarths` 的其它相关初始化值进行设置，其代码如下，：

```

uint32_t freq = k210_get_clk_freq();
uint16_t div = freq / 115200 - 1;
/* Set UART registers */
uarths->div.div = div;
uarths->txctrl.txen = 1;
uarths->rxctrl.rxen = 1;
uarths->txctrl.txcnt = 0;
uarths->rxctrl.rxcnt = 0;
uarths->ip.txwm = 1;
uarths->ip.rxwm = 1;
uarths->ie.txwm = 0;
uarths->ie.rxwm = 0;

```

初始化结束，则可以根据相应 `uart_puts()` 函数或者 `printf()` 函数通过串口输出调试信息。可以看到系统输出 `Hello,RVOS!`，这表明串口信息输出正常。各设备测试结果如下：

```

root@hecs-33671:~/RVOS/Code# make run
Makefile:43: warning: overriding recipe for target 'clean'
Makefile:21: warning: ignoring old recipe for target 'clean'
Press Ctrl-A and then X to exit QEMU
-----
Hello,RVOS!

```

图 5.4 QEMU-Virt 设备初始化测试结果

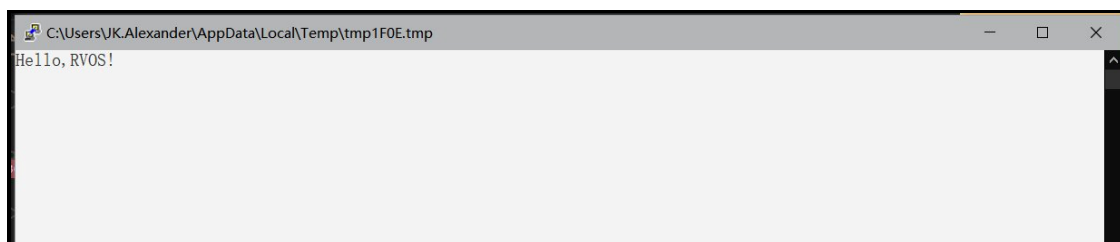


图 5.5 K210 初始化测试结果

5.3 内存管理测试

5.3.1 page 级内存管理

page 级内存管理主要测试集中在其是否合理创建与释放上，测试代码如下：

```

void *p = page_alloc(2);
printf("p = 0x%x\n", p);
void *p2 = page_alloc(7);
printf("p2 = 0x%x\n", p2);
page_free(p2);
void *p3 = page_alloc(4);
printf("p3 = 0x%x\n", p3);

```

测试时，程序首先会创建内存块 p ，其大小为 2 页，之后，创建内存块 $p2$ ，其大小为 7 页，在 $p2$ 创建后，将其释放。之后创建内存块 $p3$ ，其大小为 4 页。测试结果如下：

```

Press Ctrl-A and then X to exit QEMU
-----
Hello, RVOS!
HEAP_START = 800033f4, HEAP_SIZE = 07ffcc0c, num of pages = 32756
TEXT: 0x80000000 -> 0x80002cec
RODATA: 0x80002cec -> 0x80002e7b
DATA: 0x80003000 -> 0x80003000
BSS: 0x80003000 -> 0x800033f4
HEAP: 0x8000c000 -> 0x88000000
p = 0x8000c000
p2 = 0x8000e000
p3 = 0x8000e000

```

图 5.6 page 级内存管理测试结果

可以看到有初始化时系统各段地址输出，同时， $p2$ 和 $p3$ 时，由于释放了 $p2$ 所以，他们两个起始位置一致。我们可以从图 5.7 中直观的看到测试过程中内存发生的变化。首先堆中出现第一块内存 p 。之后， $p2$ 被创建，大小为 7。此时将其释放，堆区可分配区域只剩内存块 p ，此时创建 $p3$ ，其位置就会紧跟内存块 p 。

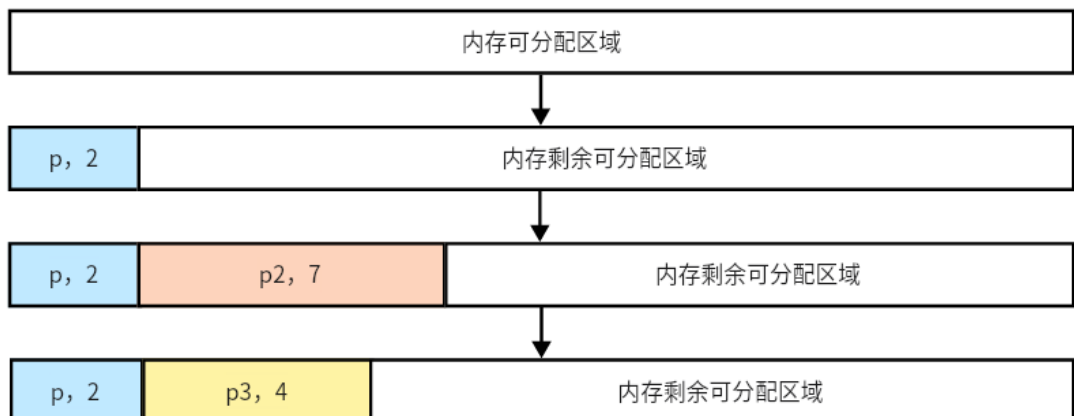


图 5.7 page 级内存管理测试结果内存变化图

5.3.2 malloc 级内存管理

malloc 级内存管理的初始化仅需建立一个最大的初始化空闲内存块,实际上,你可以将剩余的内存分配是在这个最大空闲内存块上的分割。

在测试中,我设计了如下代码:

```
void *p = malloc(1024);
printf("p = 0x%lx\n", p);
void *p1 = malloc(512);
printf("p1 = 0x%lx\n", p1);
void *p2 = malloc(256);
printf("p2 = 0x%lx\n", p2);
void *p3 = malloc(256);
printf("p3 = 0x%lx\n", p3);
void *p4 = malloc(128);
printf("p4 = 0x%lx\n", p4);
free(p1);
free(p3);
free(p2);
void *p5 = malloc(1040);
printf("p5 = 0x%lx\n", p5);
```

可以看到,首先我分配了 5 块内存 p/p1/p2/p3/p4。之后,分别释放 p1/p3/p2 来测试 free 函数的合并能力,之后,创建 p5 来查看空间是否释放。

分别对 QEMU-Virt 设备以及 K210 设备进行测试,结果如下:

```
DATA: 0x0000000000005000 -> 0x0000000000005000
BSS: 0x0000000000005000 -> 0x000000000000121c0
HEAP: 0x000000000000121c0 -> 0x00000000000060000

block_head: 0x000000000000121c0
block_size: 1024
p = 0x000000000000121d8
block_head: 0x000000000000125d8
block_size: 512
p1 = 0x000000000000125f0
block_head: 0x000000000000127f0
block_size: 256
p2 = 0x00000000000012808
block_head: 0x00000000000012908
block_size: 256
p3 = 0x00000000000012920
block_head: 0x00000000000012a20
block_size: 128
p4 = 0x00000000000012a38

block: 0x000000000000125d8
front_block: 0x000000000000121c0
next_block: 0x000000000000127f0

block: 0x00000000000012908
front_block: 0x000000000000127f0
next_block: 0x00000000000012a20

block: 0x000000000000127f0
front_block: 0x000000000000125d8
next_block: 0x00000000000012908

block_size: 256 ,next_block_size: 256
block = 0x000000000000127f0, block->next = 0x00000000000012a20, new_block_size = 536
front_block_size: 512 ,block_size: 536
front_block = 0x000000000000125d8, front_block->next = 0x00000000000012a20, new_front_block_size = 1072

block_head: 0x000000000000125d8
block_size: 1040
p5 = 0x000000000000125f0
```

图 5.8 QEMU-Virt malloc 测试结果

```

Hello, RVOS!
num_sizes: 6217256
TEXT: 0x0000000080000000 -> 0x0000000080003fe8
RODATA: 0x0000000080003fe8 -> 0x00000000800045e0
DATA: 0x0000000080005000 -> 0x0000000080005000
BSS: 0x0000000080005000 -> 0x00000000800121c0
HEAP: 0x00000000800121c0 -> 0x0000000080600000

block_head: 0x00000000800121c0
block_size: 1024
p = 0x00000000800121d8
block_head: 0x00000000800125d8
block_size: 512
p1 = 0x00000000800125f0
block_head: 0x00000000800127f0
block_size: 256
p2 = 0x0000000080012808
block_head: 0x0000000080012908
block_size: 256
p3 = 0x0000000080012920
block_head: 0x0000000080012a20
block_size: 128
p4 = 0x0000000080012a38

block: 0x00000000800125d8
front_block: 0x00000000800121c0
next_block: 0x00000000800127f0

block: 0x0000000080012908
front_block: 0x00000000800127f0
next_block: 0x0000000080012a20

block: 0x00000000800127f0
front_block: 0x00000000800125d8
next_block: 0x0000000080012908

block_size: 256 ,next_block_size: 256
block = 0x00000000800127f0, block->next = 0x0000000080012a20, new_block_size = 536
front_block_size: 512 ,block_size: 536
front_block = 0x00000000800125d8, front_block->next = 0x0000000080012a20, new_front_block_size = 1072

block_head: 0x00000000800125d8
block_size: 1040
p5 = 0x00000000800125f0

```

图 5.9 K210 malloc 测试结果

可以看到，在分配完五个内存块后，p1 与 p3 的释放由于前后无空闲内存块，所以没有造成内存块合并，而 p2 释放时，前后有 p1 与 p3 两个空闲块，所以进行了合并，之后，p5 在创建时，由于中间的内存块足够大，所以，他的起始位置与 p1 相同。用内存变化图可以更加直观的表述这一变化，首先是创建过程，如下：

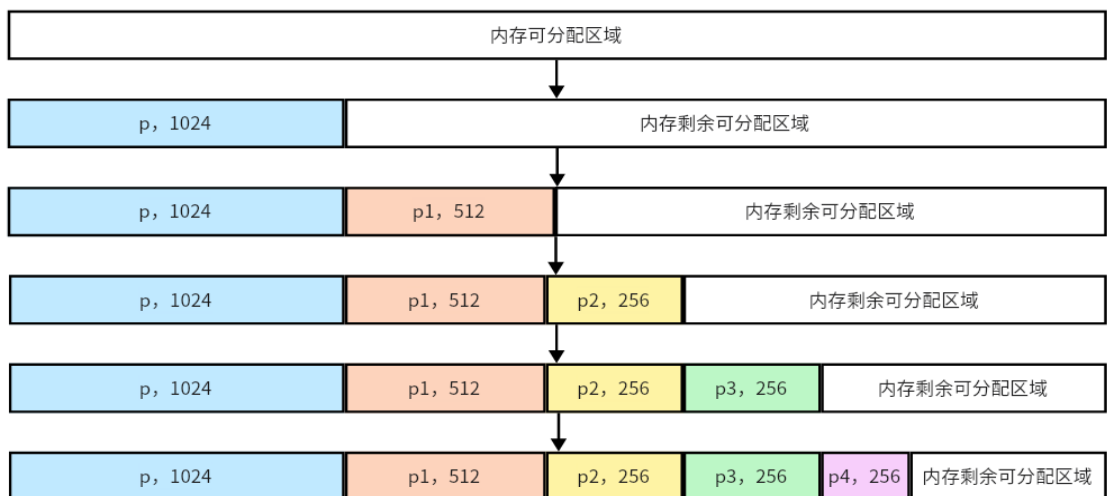


图 5.10 malloc 内存管理结果内存分配图

创建时五块内存依次存放到内存中。同样，回收时如下所示。首先 p1 被释放，

由于前后无内存空闲块，没有进行合并。之后，释放 p3 同样由于前后无空闲内存块，没有进行合并，最后释放内存块 p2，此时前后都有空闲的内存块，所以进行了空闲块合并。中间出现一个较大的空闲区域。

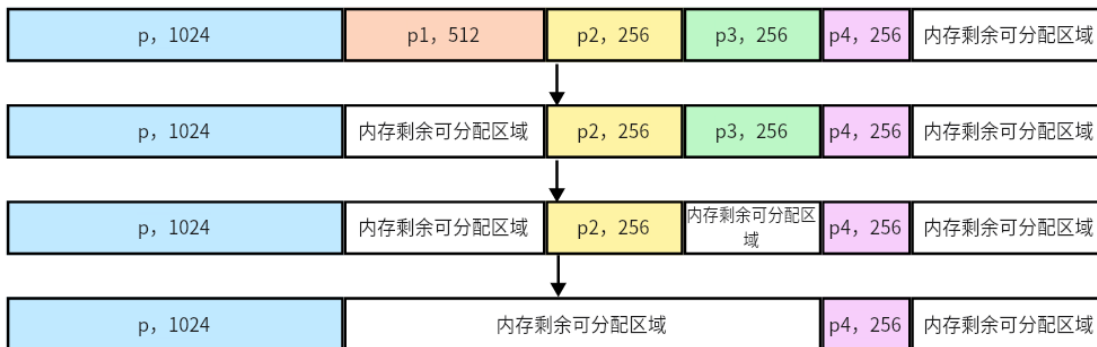


图 5.11 malloc 内存管理结果内存回收图

5.4 任务调度模块测试

RVOS 的任务调度是基于优先级的协作式任务调度。在测试时，我设计的方式是创建三个任务，其中，有两个任务优先级最高为 0，另外一个任务优先级稍低为 1，每个任务都会打印 runing。测试代码如下：

```
task_create(user_task0,NULL,0);
task_create(user_task1,NULL,0);
task_create(user_task2,NULL,1);
```

创建完成后，任务队列中的任务如图 5.12 所示。由于 task0 与 task1 的优先级皆为 0，二者形成了一个同一优先级的双向链表。task2 的优先级为 1，它只的链表指针首尾相接。

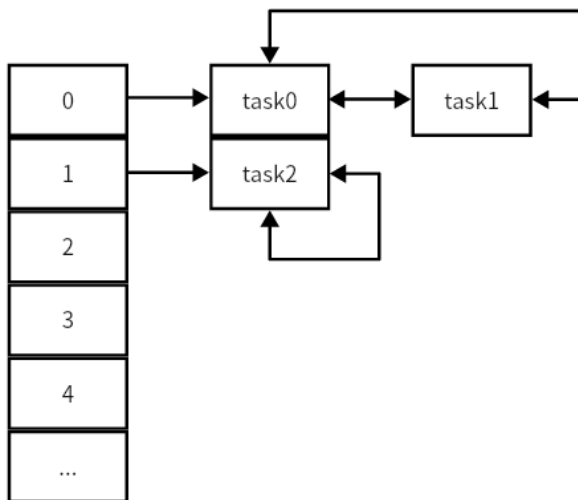


图 5.12 任务调度初始任务分布图

测试结果如图 5.13 和图 5.14 所示。可以看到，开始三个任务创建后，由于优先级高，task0 与 task1 交替运行 5 次，之后 task0 自动释放，此时 task1 优先级比 task2 高，task1 又运行 5 次，之后释放，task2 开始一直运行。

```
block_head: 0x0000000080012ab8  
block_size: 4368  
block_head: 0x0000000080013be0  
block_size: 4368  
block_head: 0x0000000080014d08  
block_size: 4368  
Task 1: Created!  
Task 1: Running...  
Task 0: Created!  
Task 0: Running...  
Task 1: Running...  
Task 0: Running...  
Task 1: Running...  
Task 0: Running...  
Task 1: Running...  
Task 0: Running...  
Task 1: Running...  
Task 0: Running...  
Task 1: Running...  
  
block: 0x0000000080012ab8  
front_block: 0x0000000080012a20  
next_block: 0x0000000080013be0  
  
Task 1: Running...  
Task 1: Running...  
Task 1: Running...  
Task 1: Running...  
  
block: 0x0000000080013be0  
front_block: 0x0000000080012ab8  
next_block: 0x0000000080014d08  
  
front_block_size: 4368 ,block_size: 4368  
front_block = 0x0000000080012ab8, front_block->next = 0x0000000080014d08, new_front_block_size = 8760  
  
Task 2: created!  
Task 2: Running...  
Task 2: Running...  
Task 2: Running...  
Task 2: Running...  
Task 2: Running...  
Task 2: Running...
```

图 5.13 OEMU-Virt 任务调度测试结果

```

block_head: 0x0000000080012ab8
block_size: 4368
block_head: 0x0000000080013be0
block_size: 4368
block_head: 0x0000000080014d08
block_size: 4368
Task 1: Created!
Task 1: Running...
Task 0: Created!
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
block: 0x0000000080012ab8
front_block: 0x0000000080012a20
next_block: 0x0000000080013be0

Task 1: Running...
Task 1: Running...
Task 1: Running...
Task 1: Running...

block: 0x0000000080013be0
front_block: 0x0000000080012ab8
next_block: 0x0000000080014d08

front_block_size: 4368 ,block_size: 4368
front_block = 0x0000000080012ab8, front_block->next = 0x0000000080014d08, new_front_block_size = 8760

Task 2: Created!
Task 2: Running...
Task 2: Running...
Task 2: Running...
Task 2: Running...
Task 2: Running...
Task 2: Running...
Task 2: Running...

```

图 5.14 K210 任务调度测试结果

5.5 Trap 处理测试

5.5.1 异常测试

异常部分我使用了一个经典的 Store/AMO access fault 作为测试，代码如下：

```
/*
 * Synchronous exception code = 7
 * Store/AMO access fault
 */
*(int *)0x0000000000000000 = 100;
```

程序输出了提示异常触发的调试信息 Sync exceptions!以及该异常的类型报错码，如图 5.15 所示。由于暂时未设计相关的处理流程，RVOS 将调用 panic 函数实现异常中止，我们可以看到 panic 函数的中止前调试信息。

```
Task 0: Created!
Task 0: Running...
Sync exceptions!, code = 7
0000000000000007
panic: OOPS! What can I do!
```

图 5.15 QEMU-Virt 异常测试结果

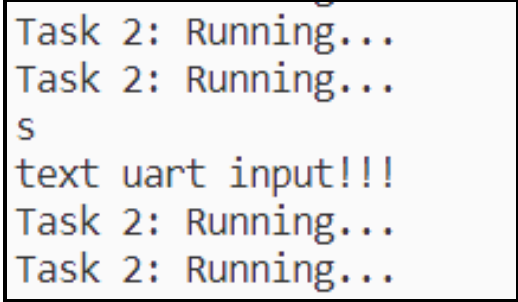
但是，基于 K210 的硬件设计，其 0 地址处运行赋值，故此处使用地址非对齐来进行测试，结果如下：

```
Task 0: Created!
Task 0: Running...
Sync exceptions!, code = 6
0000000000000006
panic: OOPS! What can I do!
```

图 5.16 K210 异常测试结果

5.5.2 中断测试

中断测试使用的是 uart 设备的中断，实现上为每当输入字符时，触发中断，测试结果如下：



```
Task 2: Running...
Task 2: Running...
s
text uart input!!!
Task 2: Running...
Task 2: Running...
```

图 5.17 QEMU-Virt 设备中断测试

可以看到，当输出字符 s 时，中断触发，在进入对应 uart 中断处理程序后，处理程序使用 `uart_getc()` 函数获取输入缓存区的字符，之后调用 `uart_putc(char c)` 函数将该字符回显，之后打印相关的调试信息。

第六章 总结与展望

操作系统作为计算机科学的结晶之一，分类十分庞大，而其一类中的实时操作系统，其功能也众多。RVOS 在设计中仅实现了一个实时操作系统最基础的几个模块，他们包括内核启动，串口调试输出，内存管理与任务调度等。但是同时，也存在一些问题尚未解决。

第一点，RVOS 的 page 级内存管理在 64 位系统中存在运行问题，由于后续使用时一直在用 malloc 级内存管理来进行内存分配，故该方面问题尚未解决。

第二点，RVOS 在移植到 K210 的过程中，其中断仍无法正常触发，分析原因是 K210 芯片的中断系统存在问题，暂时尚未解决，造成 K210 的中断系统尚未完整建立。

展望未来，RVOS 将会不断进行迭代性开发，主要的规划在以下几点：

第一，对于内存管理部分，RVOS 未来将结合 Linux 的设计，融合 page 级内存管理与 malloc 级内存管理的优点，使用伙伴系统来管理内存，这将使内存的创建与回收更加快捷，同时进一步减少内存碎片化的趋势。同时，我也会开始设计虚拟内存，最终实现虚拟内存机制。

第二，对于任务调度部分，RVOS 未来将实现定时器部分，之后利用定时器、中断等部分，将协作式任务调度生成为抢占式任务调度，这将使 RVOS 支持更多的功能。

第三，对于一个现代操作系统，系统调用是不可或缺的存在，RVOS 未来计划设计 Machine 层与 User 层双层架构，使 RVOS 的安全性进一步提高。

第四，在交互性设计上，RVOS 现在仍十分缺失，所以未来，RVOS 将设计更多的交互式应用，使其支持命令行指令。

第五，未来，RVOS 将添加完整的文件系统，使其支持更多功能。

第六，在移植性上，RVOS 将在 K210 上实现更多的外设驱动，来扩展 RVOS 的功能，同时，也将扩展更多的开发板。

在完成毕设之后，RVOS 仍将作为一个长期项目来持续完整，项目目前已开源到 github 上作为一个开源项目存在。

致 谢

首先，感谢李航老师在大二时允许我加入了他的实验室，是他引领我走上了嵌入式这条道路，没有李航老师，可能我整个大学的时光将荒废大半。

其次，感谢张剑贤老师在毕设期间的指导，张剑贤老师工作认真细致，既能和我对一个问题讨论许久，也会对我的工作提供非常多的帮助。

然后，感谢技术交流群的朋友们，如果没有你们，我可能很难解决毕设过程遇到的好多技术问题。

最后，感谢亲爱的爸爸妈妈，可能大学四年我并没有完美的满足你们的期望，但是，你们始终如一的支持我，始终对我的选择予以尊重。

参考文献

- [1]. Tang, Yu, et al. "A comprehensive survey of RISC-V hardware and software ecosystem." *Journal of Systems Architecture* 101 (2019): 101674.
- [2]. 马威, 姚静波, 常永胜, 解维奇. 国产 CPU 发展的现状与展望[J]. *集成电路应用*, 2019,36(04):5-8.
- [3]. 雷思磊. RISC-V 架构的开源处理器及 SoC 研究综述[J]. *单片机与嵌入式系统应用*, 2017,17(02):56-60+76.
- [4]. 何小庆. RISC-V 处理器嵌入式开发概述[J]. *单片机与嵌入式系统应用*, 2020,20(11):1-6.
- [5]. KIM M. Unlocking JavaScript: V8-RISCV Open Sourced[EB/OL]. [2021-05- 05]. <https://riscv.org/blog/2020/08/unlocking-javascript-v8-riscv-open-sourced/>.
- [6]. LABROSSE J. 嵌入式实时操作系统 $\mu C\backslash OS-II$ [M]. 邵贝贝, 译. 第 2 版. 北京: 北京航空航天大学出版社, 2003.
- [7]. 何小庆. 3 种物联网操作系统分析与比较[J]. *微纳电子与智能制造*, 2020,2(01):65-72.
- [8]. 胡振波. RISC V 架构与嵌入式开发快速入门[M]. 北京: 人民邮电出版社, 2018:34 39.
- [9]. WATERMAN A, ASANOVI' C K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213[S]. RISC-V Foundation, 2019.
- [10]. WATERMAN A, ASANOVI' C K. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSURatified[S]. RISC-V Foundation, 2019.
- [11]. 胡振波. (2018). RISC V 架构与嵌入式开发快速入门 [M]. 北京. 人民邮电出版社, 2018:34-39.
- [12]. Cyril Bresch;David Hely;Stéphanie Chollet;Roman Lysecky;Ioannis Parissis.TrustFlow-X: A Practical Framework for Fine-Grained Data Flow Integrity in Critical Systems[J]. *ACM Transactions on Embedded Computing Systems*, 2020
- [13]. Sajjad Tamimi;Zahra Ebrahimi;Behnam Khaleghi;Hossein Asadi.An Efficient SRAM-Based Reconfigurable Architecture for Embedded Processors[J]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019(3)
- [14]. 冯建文. 基于 RISC-V 架构的中断实验设计[J]. *实验室研究与探索*, 2022(12)
- [15]. EE Times. 2019 Embedded Markets Study[R]. San Francisco: EE Times, 2019.