

上海应用技术大学

毕业设计(论文)

课题名称: 基于 Risc-V 的操作系统内核
模拟设计与实现

学 院 计算机科学与信息工程学院

专 业 软件工程

班 级 19104221 学号 1910400740

学生姓名 张智强

指导教师 张成姝

起止日期 2022.12.23—2023.5.5

上海应用技术大学毕业设计（论文）任务书

题目：基于 Risc-V 的操作系统内核模拟设计与实现	
学生姓名：张智强 学号：1910400740 专业：软件工程	
任务起至日期：2022 年 12 月 23 日至 2023 年 5 月 5 日 共 16 周	
<p>一、课题的任务内容：</p> <ol style="list-style-type: none"> 1. 通过毕业设计，了解和参与从软件工程角度出发设计一个系统的各个步骤及具体实施的基本过程； 2. 能熟练运用一种编程语言及熟练使用数据库管理系统进行程序设计；对编程能力有一定要求。 3. 主要完成以下功能：在学习了解 RISC-V 架构以及相关集成工具的基础上，模拟设计操作系统的微内核，提供必要的服务，包括进程管理，内存管理，文件系统等。编程实现各模块基本功能，如在进程管理中实现进程的创建，进程的调度算法。 	
<p>二、原始条件及数据：</p> <ol style="list-style-type: none"> 1. 对各功能模块有初步划分； 2. 开题报告、毕业论文模板及相关参考文档。 	
<p>三、设计的技术要求（论文的研究要求）：</p> <p>本课题要求模拟设计和实现操作系统部分微内核功能，主要内容包括：</p> <ol style="list-style-type: none"> 1. 进程管理：编程实现创建进程，进程的调度算法等。 2. 内存管理：编程实现内存分配和回收算法。 3. 文件系统：提供操作系统内文件的管理。 	
<p>四、毕业设计（论文）应完成的具体工作：</p> <ol style="list-style-type: none"> 1. 设计和实现系统的各项主要功能； 2. 撰写毕业论文，毕业论文中应该包括课题说明、系统的详细功能需求、系统总体结构设计、系统模块设计、数据库设计、基本对象设计、程序开发过程说明、使用调试过程说明等。 <p>软硬件名称、内容及主要的技术指标（可按以下类型选择）：</p>	
计算机软件	设计和开发系统，达到可以演示运行的使用效果。
图 纸	
电 路 板	
机 电 装 置	
新材料制剂	
结 构 模 型	
其 他	

五、查阅文献要求及主要的参考文献:

- [1] O. M. Ritchie and K. Thompson, "The UNIX time-sharing system," in The Bell System Technical Journal, vol. 57, no. 6, pp. 1905-1929, July-Aug. 1978, doi: 10.1002/j.1538-7305.1978.tb02136.x.
- [2] Bhat W A, Quadri S M K. Review of FAT Data Structure of FAT32 file system[J]. Oriental Journal of Computer Science & Technology, 2010, 3(1): 161-164.
- [3] Leijen D, Zorn B, de Moura L. Mimalloc: Free list sharding in action[C]//Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1 – 4, 2019, Proceedings 17. Springer International Publishing, 2019: 244-265.
- [4] Cox R, Kaashoek M F, Morris R. Xv6, a simple Unix-like teaching operating system[J]. 2022-11-05]. <https://pdos.csail.mit.edu/6.1810/2022/xv6/book-riscv-rev3.pdf>, 2022.
- [5] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1", Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 31, 2016.
- [6] 陈海波. 现代操作系统: 原理与实现[M]. 机械工业出版社, 2020.
- [7] 张成姝. 操作系统教程 (第二版) [M]. 清华大学出版社, 2019.
- [8] 曹成. 嵌入式实时操作系统 RT-Thread 原理分析与应用[D]. 济南: 山东科技大学, 2011.
- [9] 邱伟. 嵌入式实时操作系统 RT-Thread 的设计与实现[D]. 电子科技大学, 2007.
- [10] 胡振波. RISC-V 架构与嵌入式开发快速入门[M]. 人民邮电出版社, 2019.
- [11] 林金龙. 深入理解 RISC-V 程序开发[M]. 北京航空航天大学出版社, 2021.
- [12] 于渊. Orange'S: 一个操作系统的实现[M]. 电子工业出版社, 2009.
- [13] 张依依. RISC-V 2023: 难点也是突破点[N]. 中国电子报, 2023-01-13(008). DOI:10.28065/n.cnki.ncdz.2023.000056.

六、进度安排: (设计或论文各阶段的要求, 时间安排):

2022—2023 年度第 1 学期 17—19 周: 接受任务, 外出调研, 书籍和资料的准备, 撰写开题报告;

2021—2022 年度第 2 学期第 1 周: 提交中英文翻译资料, 系统分析, 明确系统各个模块的功能;

第 2 周至 6 周: 程序设计; 模块调试;

第 7 周至 10 周: 完善系统, 撰写论文, 答辩准备;

指导教师: 张成姝

2022 年 12 月 10 日

审核意见: 同意

教研室主任: 荣祺

2021 年 12 月 20 日

基于 Risc-V 的操作系统内核模拟设计与实现

摘要： 本课题基于 RISC-V 处理器架构设计并实现一个简易的以教学为主要目的的嵌入式操作系统内核，按照软件工程的思想进行了需求分析，概要程序设计，详细程序设计等方面进行组织。本课题的出发点是由于传统的商业化的大型操作系统内核过于复杂不便于教学，主流的 x86 架构也具有沉重的历史包袱，基于现代模块化的 RISC-V 架构平台设计的嵌入式操作系统，既抛弃了 x86 沉重的历史包袱，又大大简化了操作系统的架构，放弃了一定的安全性、高性能、可扩展性，但这使得代码结构对于操作系统初学者接触操作系统的实现相当友好。本课题完成了操作系统的引导、UART 串口驱动、中断管理、动态内存管理、多任务调度、自旋锁、信号量、FAT32 文件系统内核模块，并基于本课题的操作系统内核模块完成了一些用户态应用程序，包括链表数据结构、命令式文件管理器、生产者消费者演示程序、内存分配器演示程序、Shell 命令式人机交互接口、走迷宫小游戏、数学表达式求解器。

关键词： 嵌入式；RISC-V；操作系统；进程调度；内存管理

Simulation Design and Implementation of Operating System Kernel Based on Risc-V

Abstract: This project is based on the RISC-V to design and implement a simple embedded operating system kernel on teaching. It is organized according to the principles of software engineering, including requirements analysis, detailed program design. Traditional operating system kernels are complex for teaching purposes, and x86 architecture carries a heavy historical burden. By using modern modular RISC-V for embedded operating systems, we can abandon the heavy historical burden and simplify the architecture of operating systems. This sacrifices some security, high performance, and scalability, but makes the code more friendly for learning about operating system implementation. This project has completed the boot process of the operating system, UART serial port driver, interrupt management, dynamic memory management, multitasking scheduling, spin lock, semaphore, and FAT32 file system kernel module. Based on the kernel module of this project's operating system, several user-space applications have been developed, including linked list data structure, command-based file manager, producer-consumer demonstration program, memory allocator demonstration program, Shell command-line interface, maze game, and mathematical expression solver.

Keywords: Embedded; RISC-V; Operating system; Process scheduling; Memory Management

目录

1 绪论	1
1.1 选题背景与意义	1
1.2 国内外研究现状	1
1.3 操作系统未来的发展趋势	2
1.4 本课题研究内容	3
2 需求分析	4
2.1 功能性需求	4
2.2 非功能性需求	5
2.3 UML 需求用例建模	5
3 内核设计与实现	10
3.1 开发环境与准备工作	11
3.2 编译与链接	12
3.3 从汇编到 C 的引导	13
3.4 串口设备初始化	15
3.5 动态内存管理	17
3.6 进程管理与调度	20
3.7 进程同步	24
3.7.1 自旋锁	24
3.7.2 信号量	26
3.8 文件系统	28
3.8.1 FatFs 项目概述与移植	29
3.8.2 磁盘分区与格式化操作	32
3.8.3 文件的创建与读写	32
3.8.4 目录创建与目录列举	33
4 用户程序案例实现	34
4.1 链表数据结构	34
4.2 生产者—消费者演示程序	35
4.3 文件系统目录的递归遍历	38
4.4 SHELL 命令操作接口的实现	40
5 总结	45
致谢	47
参考文献	48
附录	49

1 绪论

1.1 选题背景与意义

目前在商用个人计算机的操作系统市场领域,已经几乎被 Windows、MacOS 所全面占据,近几年随着软件国产化的倡导,开源的 Linux 桌面平台也在逐步进入人们的视野。移动端操作系统的代表几乎只剩下了 Android、iOS。为了研究一个真实计算机操作系统的设计与实现, Linux 这个开源的操作系统内核可以充当一个操作系统领域一篇精彩的案例,然而 Linux 从上世纪 90 年代发展至今已经成为了具有 2000 多万行代码量的庞然大物,已经几乎不可能是个人能够彻底研究清楚的工程了。

目前大多数操作系统的案例仍然是基于传统的 x86 指令集架构所开发,得益于 CISC 可变长指令的易扩展性, x86 从上世纪 70 年代的 Intel 8086 处理器开始发展至今仍然能够兼容旧指令集,这导致其历史包袱尤为严重,若想研究 x86 体系结构下的操作系统原理,既需要学习实模式下的汇编,又需要学习保护模式下的汇编,且实模式下的汇编有些规定放在如今看来已经是一些过时的设计。这导致其汇编指令对初学者而言相当不友好,指令繁多,每条指令有各种寻址模式,还需要了解一些历史原因知道有些特殊设计背后的原因。

RISC-V 指令集架构是一套全新的、开源的、先进的、模块化的指令集架构。RISC-V 的指令集手册在当前版本的非特权指令集由 238 页,特权指令集有 155 页,相比于 x86 的几千页指令集手册可谓是相当的简洁。

由于 Linux 和 x86 的复杂性与历史包袱,故模拟设计与实现一个采用现代 RISC-V 架构的以教学为目的的操作系统就显得尤为重要。

1.2 国内外研究现状

目前商业主流操作系统有 Windows、Linux、MacOS、Android、iOS,这些都是商用的已经拥有成熟生态的大型操作系统。本课题的研究目标以教学使用为目的,需要使用尽量小巧,简洁,清晰的代码完成一个操作系统的核心功能,而并不需要足够完善,覆盖各种用户场景,故这类大型商业化的操作系统并不是本研究课题所探讨的范围。对于小型而简洁的操作系统,实际上在嵌入式操作系统领域内,已经拥有相当多的精彩的可供参考的案例了。

FreeRTOS 是完全免费开源的嵌入式操作系统,可灵活裁剪,调整调度策略,方便移植到各类嵌入式设备上运行,它包括了任务管理,时间管理,信号量机制,消息队列,内存管理等,满足了大多数场景下的需求。

LiteOS 是华为针对物联网领域推出的一款嵌入式操作系统,为开发者提供了“一站

式”完整的软件平台，大大降低了物联网开发的门槛，缩短了开发周期。

RT-Thread 是由国人自主研发，国内最成熟，最完善的开源 RTOS。拥有了操作系统所需的几乎所有核心组件，并且拥有庞大的开源社区的支持。

以上列举的几个嵌入式操作系统均为目前市面上流行的嵌入式操作系统。这些嵌入式操作系统虽然相比于真正的大型操作系统已经足够简单小巧了，但是由于其目标依然是以工程实用性为目的的，为了保证良好的通用型、兼容性与可移植性，依然引入了相当多的非核心功能的实现代码和抽象，不利于用于以教学为目的进行参考。

1.3 操作系统未来的发展趋势

(1) 分布式操作系统

计算机操作系统是控制与管理操作系统的软硬件资源，合理组织与调度计算机工作与资源的分配。广义上讲，它并不仅仅是局限于单台计算机上，单机的潜能注定有限，未来的发展方向是将多个分布在各处、可随时离线（不可靠的）的计算机共同协同进行统一资源调配，实现一个逻辑上可靠的分布式系统。例如，大数据领域中的 Hadoop 框架也可称作为一种“操作系统”，它提供了 MapReduce 计算引擎，可将一个任务拆分为多个任务调度到不同的计算机上处理运行，提供了分布式文件系统（HDFS）允许各个节点进行统一访问文件存储，提供了 Yarn 进行统一的资源调度。用户提交到 Hadoop 系统上的 MapReduce 程序执行时需要调用的 Hadoop 提供的各种服务接口实际上均可看作是某种“系统调用”。

(2) 数据持久化

文件系统是操作系统提供的一个管理磁盘的手段，可为上层应用程序提供数据持久化的服务。文件系统本质是操作系统基于物理存储设备的存储结构而建立出的一个具有分级目录层次和多文件的逻辑数据结构的抽象，通过系统调用来操作文件系统这个逻辑数据结构，每个文件相当于就是一个虚拟磁盘。实际上引入文件系统概念的最终目的只是为了方便地进行数据的持久化，但是本质上将这并不一定局限于文件，目录等概念，目录和文件仅仅是传统的文件系统抽象出的一种概念。如今数据库领域发展地相当火热，传统的关系型数据库适合存储结构化数据，具有较强的一致性约束，Key/Value 数据库易于分布式存储与易于并发访问，文档型数据库更加方便存储非结构化数据，图数据库更方便实现大数据时代下的推荐系统，数据库系统本质上也是在磁盘上去构建一种数据结构，也需要负责维护崩溃一致性与崩溃恢复，其职责依然和文件系统相同，但是提供了更丰富，更灵活，更细粒度，更加强大的数据的持久化与处理服务。

(3) 云操作系统

云操作系统是随着云计算的发展而孕育出的一个操作系统，它背后管理着海量的基础设施硬件，统筹所有软硬件资源进行资源的池化，在逻辑上将其整合成为一台服务器，为云应用程序提供一个统一标准的接口调用。用户开发云应用程序通过网络按需从云操

作系统上申请获取相应的资源，并按量计费，接入即用。它拥有海量资源、高安全性、高可用、高可靠，低成本等特性。

(4) RISC-V

RISC-V 是第五代的 RISC 指令集架构，它于 2010 年诞生于加州大学伯克利分校，很快这一款开源的 RISC-V 架构便引起了学术界和工业界的关注，作为一款开源的指令集，它打破现有的 x86 与 arm 架构的垄断地位，自诞生以来，相关的学术论文爆发性增长。RISC-V 基金会由硅谷相关的公司于 2015 年成立，至今已有超过 500 个成员，其中也包括了中国的阿里，华为等知名企业。随着各类设备的智能化与物联网化，面向各类嵌入式设备芯片行业的发展显得越来越重要，上海经济和信息化委员会于 2018 年出台了国内首个 RISC-V 扶持政策，主要方向在于面向物联网和智能终端领域。阿里平头哥随后于 2019 年发布了首款高性能的玄铁 910 RISC-V 处理器。随着近几年中国芯片受到西方的打压封锁，更有可能倒逼芯片进行国产化替代，这是一次国内和欧美等发达国家在同一起跑线上竞争的一次机会，我国很有可能在这一方向上实现弯道超车。

1.4 本课题研究内容

本研究课题需要从软件工程的角度完成研究一个操作系统的实现所需的基本组成结构和各个部分细节，计划将研究硬件资源管理、进程管理与进程同步、内存管理、文件系统持久化方面的模拟实现。实际上，本课题实现的操作系统并不能作为真正实用性的操作系统，一切功能模块的实现仅仅只做功能性需求的实现，放弃了高性能、稳定性、安全性等非功能性需求，这是为了使得操作系统的实现架构与代码更加清晰简洁，使之适用于教学。

2 需求分析

本课题将从功能性需求和非功能性需求两方面进行软件需求分析。功能性需求是指本课题需要完成的操作系统具体具备的功能和行为，是最需要关注的需求部分，将从硬件资源管理、并发任务、内存管理、文件系统等方面进行描述。非功能性需求在本课题是指与要实现的操作系统功能无关的需求，将从代码的可读性、架构简洁性、完整性等方面进行描述。

2.1 功能性需求

(1) 硬件资源管理

设计出一个操作系统最起码需要能够直观地看出它在正常工作，本课题计划使用 UART 串口设备作为操作系统人机交互的直接输入输出设备，故需要编写 UART 串口设备驱动。CPU 硬件资源将通过进程管理和调度来得到有效利用。物理内存资源的管理将实现一个动态内存分配器，实现对指定物理内存区域的动态管理分配与释放。块设备的管理将实现块设备的读写驱动，在此基础上完成 FAT32 文件系统，实现文件级别的数据读写。中断管理，编写中断控制器的驱动代码与中断处理程序，实现操作系统的中断管理，以处理某些硬件外设的中断信号，如 UART 串口输入的中断，定时器中断等。

(2) 并发任务

一个操作系统基本都要至少实现并发任务的控制，需要设计一组进程管理的接口来实现进程创建与销毁功能，设计一些供进程同步使用的原语，设计实现进程调度算法，如按优先级调度，按时间片轮转策略调度等，确保操作系统能够有效地进行多任务调度。需要设计实现一些用于进程间通信机制的原语，如信号量，互斥锁等，来实现进程同步。最终实现生产者—消费者模型的操作系统课程经典的并发案例。

(3) 内存管理

实现栈内存管理与堆内存管理。栈内存主要通过汇编初始化代码初始化好固定区域的栈空间后，编译器将自动管理该区域内存的使用与回收。堆内存则需要自己设计数据结构并实现一些内存分配算法。由于本课题目标为以教学为目的的简易的嵌入式操作系统，故不引入较为复杂的虚拟内存映射机制，将直接访问指定区域的物理内存作为全局的堆内存区。最终将完成一个链表数据结构的案例。

(4) 文件系统持久化

文件系统也是操作系统组成的一部分，它负责对用户上层提供以文件的方式管理存储设备的一层抽象，本课题计划编写磁盘驱动实现磁盘设备的读写并通过自行编写或引入第三方 FAT32 文件系统驱动，最终实现目录创建，目录列举，文件读写等操作。最终完成实现一个综合性的文件系统目录树遍历的案例。

2.2 非功能性需求

(1) 简洁性

考虑到本课题所研究的嵌入式操作系统仅仅以教学为目的，所有功能并不需要考虑实际的应用场景，故所有功能性需求的实现仅作为基本的核心实现，即使这在实际软件开发中可能不利于软件的可维护性与重用性，但这能够尽可能地保持软件架构的简洁性与直观性，便于在教学中使用。

(2) 易读性

与简洁性相同，出于教学目的考虑，编写代码过程中将尽可能的不用或少用一些花哨的代码技巧，来最大可能的保持代码的易读性。

(3) 完整性

作为一个教学使用的操作系统，即使各个功能模块设计的需要保持尽可能的简单，但是由于操作系统作为一个系统性项目，各个功能模块并不是完全割裂的，很多模块的功能实现需要依赖另一个模块来提供服务，故部分模块依然需要完整实现出该模块所需的必要功能。同时，出于教学目的考虑，部分模块需要保持一定的完整性，但为了保持简洁可读性，可选择放弃一定的可维护性与可移植性。

2.3 UML 需求用例建模

在本课题研究的操作系统中，共存在着四种参与者，分别为应用层程序的开发者，编译器、链接器、操作系统内核，如图 2.1 所示。其中，开发者，编译器，链接器的参与时期在编译期，操作系统内核的参与时期在运行期。

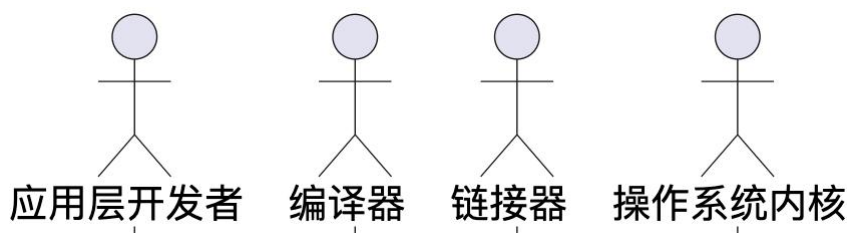


图 2.1 参与者

不同的参与者具有不同的用例，嵌入式操作系统的用户通常为嵌入式软件开发者，他们将基于嵌入式操作系统提供的各种服务完成自己的产品业务需求。

本课题计划开发者能够调用操作系统提供的进程管理接口实现多任务管理，使用文件系统管理接口实现基于文件的存储设备管理，使用进程间通信原语实现进程同步，使用动态内存分配接口实现堆内存的动态分配。同时操作系统还将负责管理一些外设设备，处理外设中断等。故可实现基本的用例图如图 2.2:

本课题计划研究的操作系统共需要实现两个外设的驱动，分别为 UART 串口驱动程序与块设备驱动程序，其中 UART 串口驱动程序用于为用户提供发送数据与接收数据的

程序接口，块设备驱动程序用于为用户提供读取与写入磁盘设备的程序接口。设备驱动在使用之前均需要完成一定的初始化工作，故可导出外设管理相关的用例图如图 2.3:

本课题计划研究操作系统中的进程管理与进程调度，应用层开发者能够通过操作系统内核提供的进程管理相关 API 函数实现创建进程，销毁进程，获取当前进程的参数，获取当前进程的 ID，阻塞当前进程（修改当前进程为阻塞态并让行 CPU），唤醒某个进程（修改当前进程为就绪态，等待被调度）。操作系统内核将在定时器中断的驱动下定时地调用进程调用入口函数，实现多任务的抢占式调度。其中，调度进程需要先按照给定的算法策略选取一个待调度的进程，要切换到该进程运行，需要切换当前 CPU 上下文，具体需要先保存当前 CPU 上下文到当前进程的 PCB 中，再恢复目标进程再 PCB 中存放的上下文数据到 CPU 中，即可成功实现上下文切换，随后更新 PCB 中存放的进程状态为运行态即可。导出用例图如图 2.4:

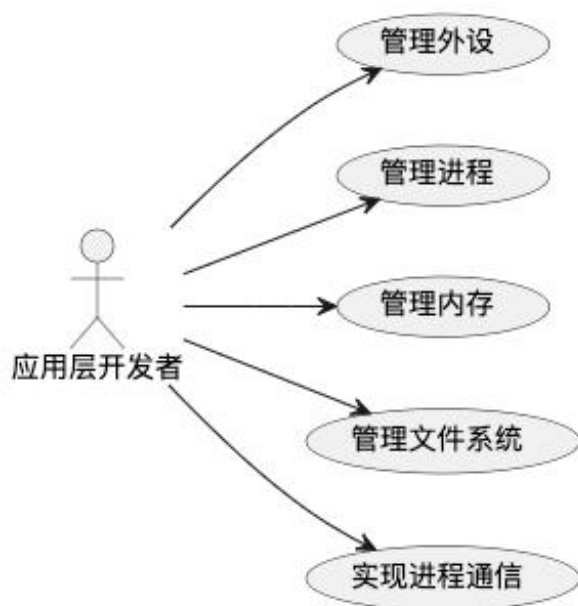


图 2.2 应用层开发者参与的用例

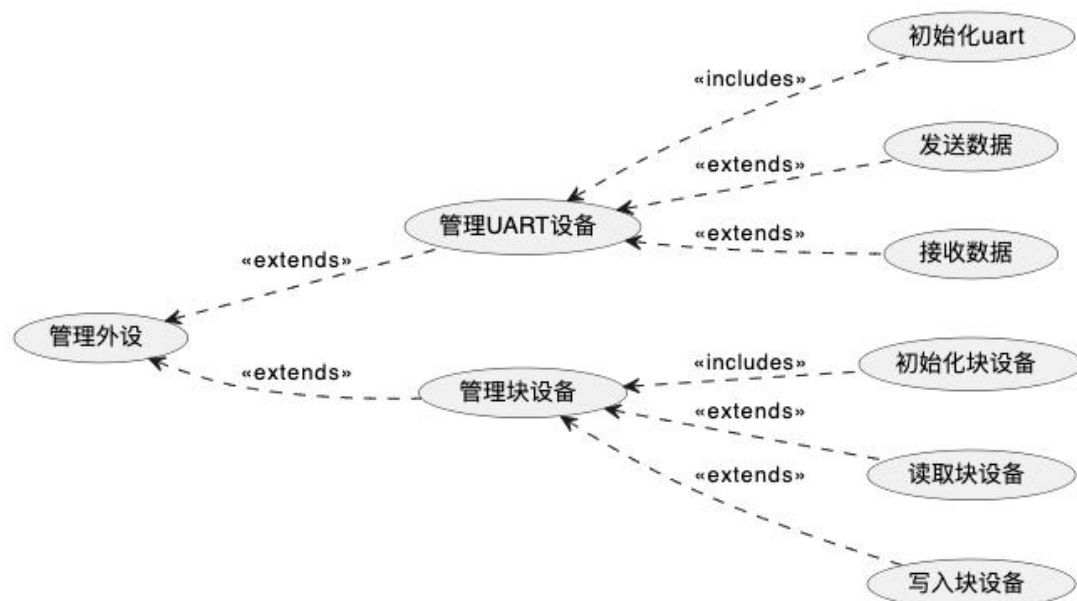


图 2.3 管理外设用例图

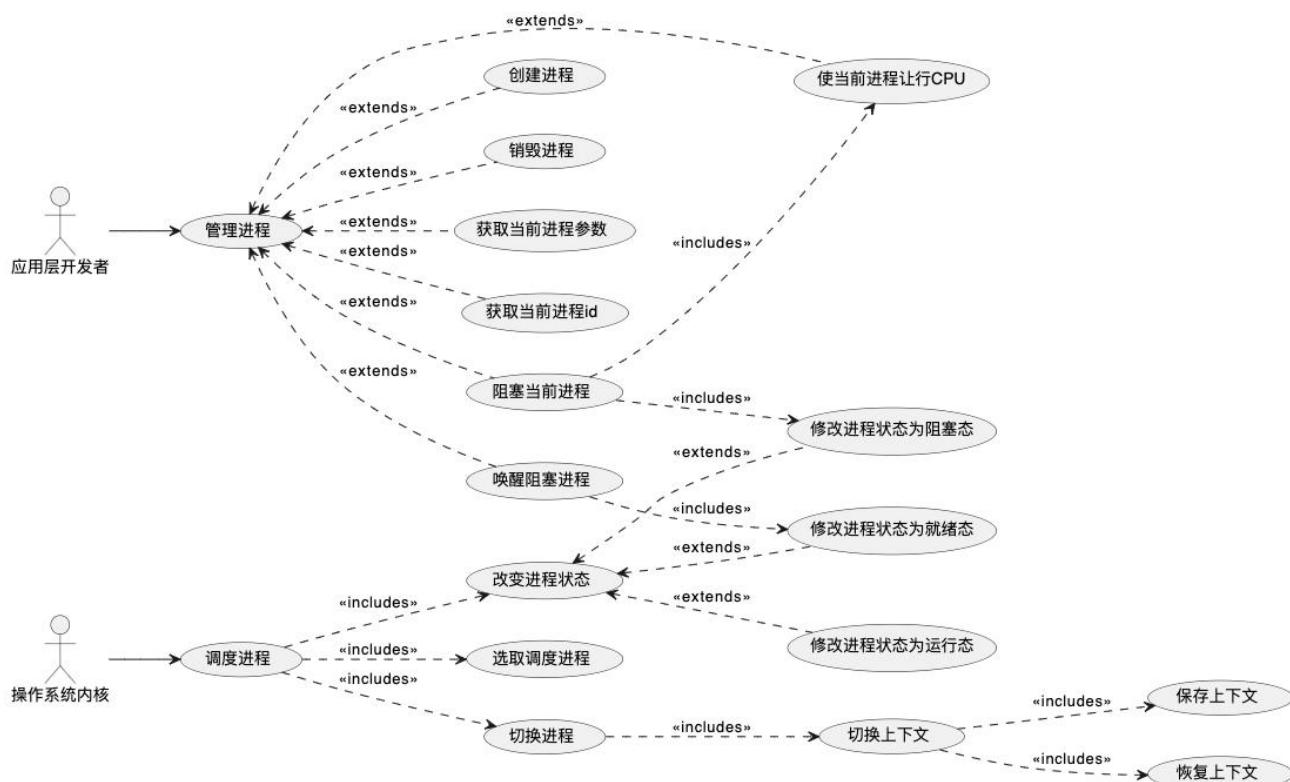


图 2.4 进程管理与调度

本课题计划实现两种进程通信的手段，分别为自旋锁与信号量。自旋锁有获取锁与释放锁两种操作，信号量有 wait，signal 两种操作。自旋锁用于保证资源能够进行互斥的访问，也可用于保证某段代码执行的原子性。信号量将基于自旋锁予以实现，它将拥有更加灵活，更加强大的进程同步的手段。导出用例图如图 2.5:

本课题计划在一个使用内存模拟实现的磁盘块设备上完成实现 FAT32 文件系统，并具备创建文件，删除文件，读取文件，写入文件，创建文件夹，删除文件夹，列举文件夹等操作。导出用例图如图 2.6:

本课题计划实现操作系统的内存管理模块，内存主要分为栈内存和堆内存，栈内存为函数调用，局部变量等存在的内存空间，由编译器自动管理，无需程序员手工干预。堆内存为开发者用户需要手动调用操作系统内核提供的内存管理相关接口完成内存空间的开辟使用与资源释放，同时堆内存所在的内存区域为程序编译后的链接阶段由链接器脚本所指定。导出用例图如图 2.7:

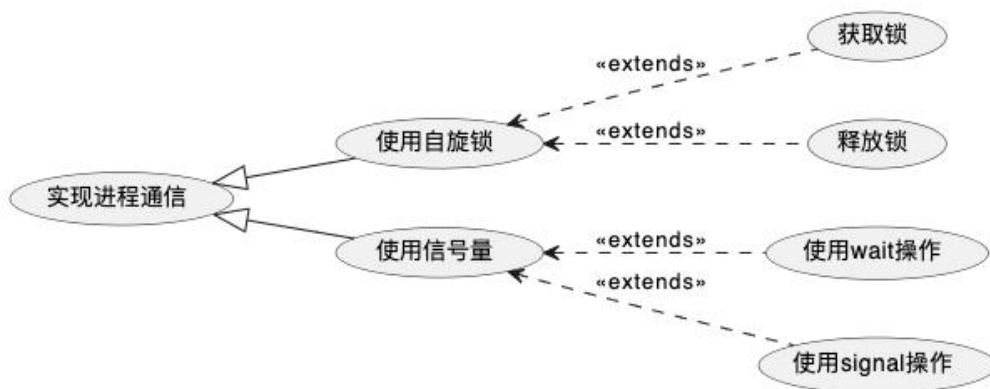


图 2.5 进程通信

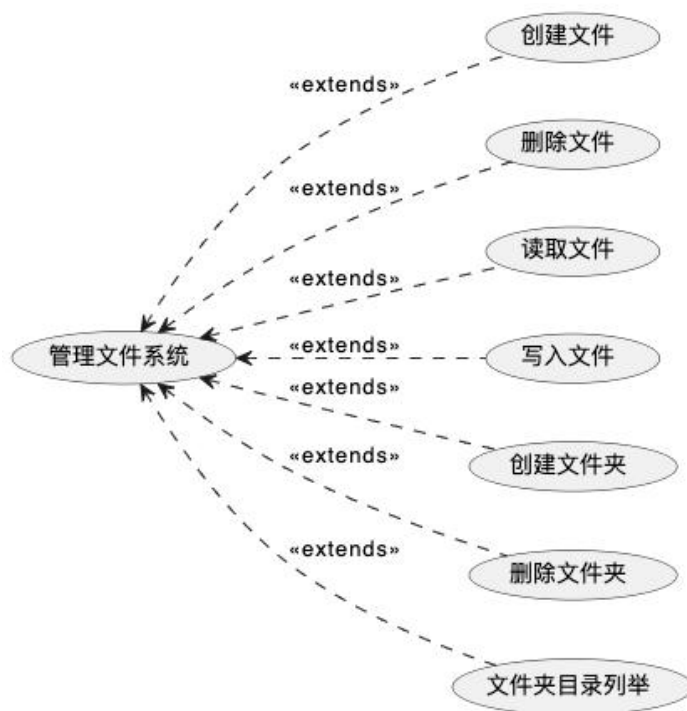


图 2.6 文件系统

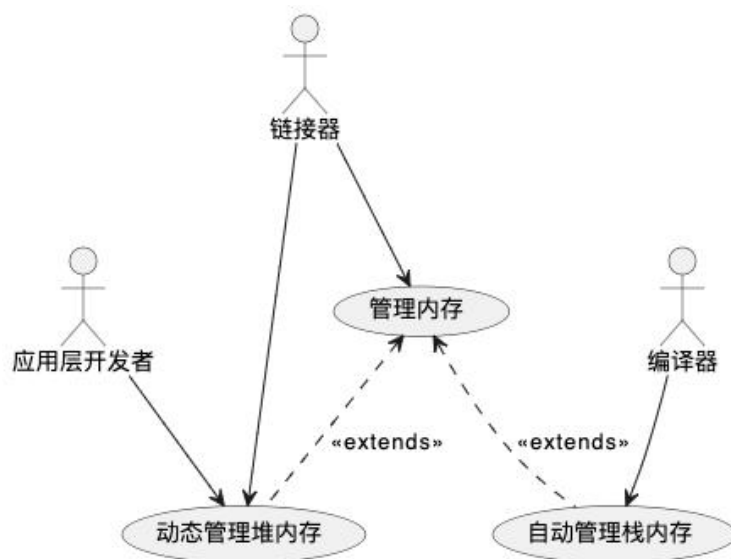


图 2.7 内存管理

3 内核设计与实现

根据需求分析阶段的用例分析，可导出操作系统的功能模块图如图 3.1:

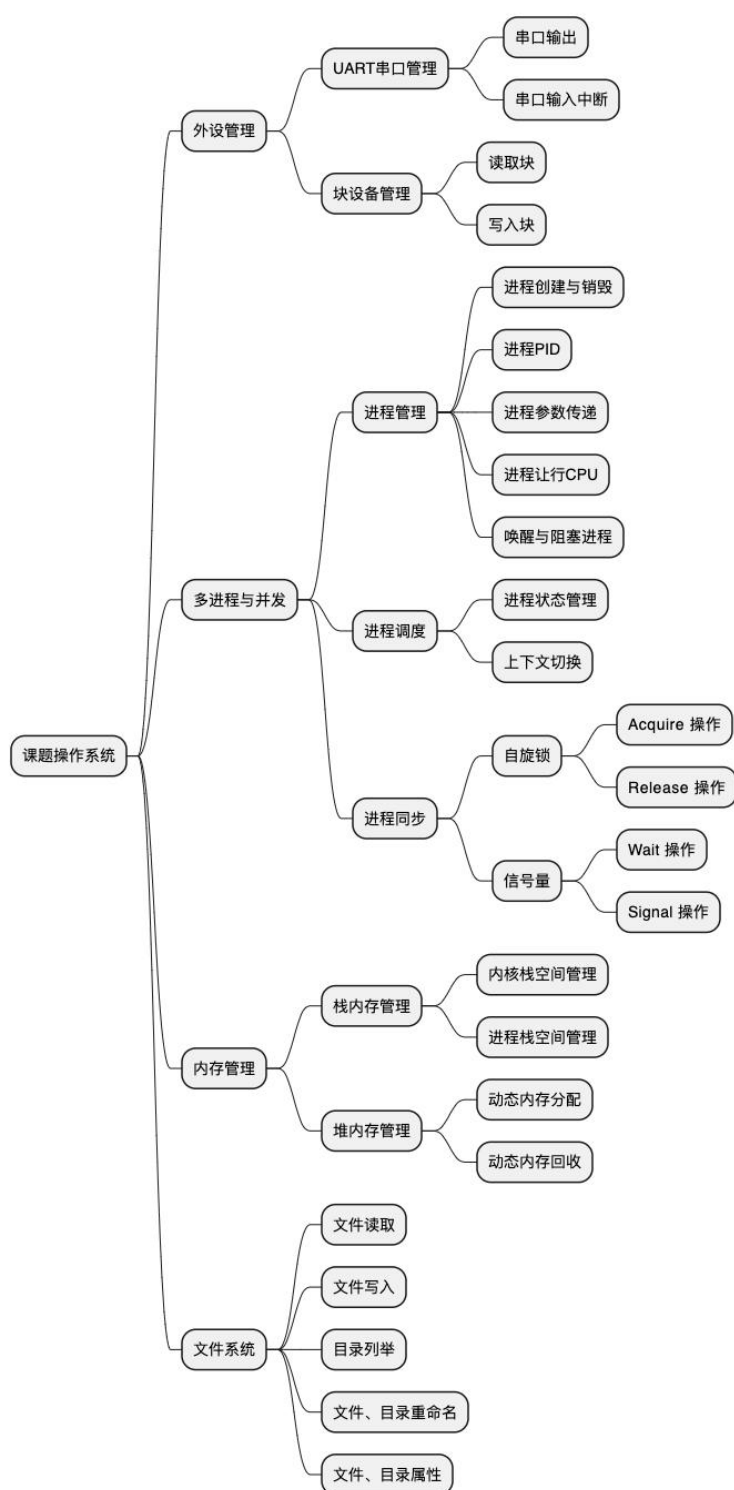


图 3.1 整体模块图

外设管理需要完成串口驱动，完成串口的输入与输出功能，完成块设备驱动的编写，实现读写磁盘块的功能。

多进程与并发部分围绕着进程展开三部分，分别为进程管理，进程调度，进程同步。进程管理包含了进程的创建与销毁、获取进程 ID、获取进程参数、进程让行、阻塞进程功能、唤醒进程功能。进程调度实现了一套抢占式的时间片轮转调度，需要依赖实现进程状态管理与进程的上下文切换功能。进程同步依靠两种工具，分别为自旋锁与信号量，自旋锁包含了 acquire 操作和 release 操作供用户程序所使用，信号量包含了 wait 操作和 signal 操作供用户程序所使用。

文件系统需要实现文件的读写，文件的重命名，目录的列举，目录的重命名，文件的属性读写操作。

内存管理分为两类内存管理，分别为栈内存管理与堆内存管理，栈内存由编译器自动完成管理，栈内存又分为了内核栈与进程栈，内核栈供内核代码所使用，每个进程都包含了一个独有的进程栈供进程专用。堆内存管理由用户程序开发者自行使用内核提供的内存分配与内存回收的操作接口来手动地完成。

3.1 开发环境与准备工作

在编写操作系统之前，需要了解目标硬件的体系结构，由于操作系统作为最底层的软件，需要直接操作裸机硬件，因此需要掌握汇编语言和 C 语言进行面向底层硬件的编程。本文将在 x86_64 架构上的 Linux 环境下进行 RISC-V 架构的程序开发，安装不同于本机架构的工具链进行编译，在嵌入式领域中这叫做“交叉编译”，本课题需要安装一系列专用于 RISC-V 架构的交叉编译工具链。

(1) GCC 编译器

本课题选用 gcc 编译器作为 C 语言编译器。在 Ubuntu 20.04 的软件源中存在着两种 RISC-V 的 GCC 编译器，分别是 gcc-riscv64-unknown-elf 和 gcc-riscv64-linux-gnu。其中，gcc-riscv64-unknown-elf 通常用于裸机的嵌入式开发，gcc-riscv64-linux-gnu 用于 RISC-V 架构的 Linux 程序的编译开发。由于本文的操作系统开发属于面向裸机的嵌入式应用开发，因此选择安装 gcc-riscv64-unknown-elf 编译器。只需要使用命令 apt install gcc-riscv64-unknown-elf 即可完成安装。

(2) Binutils 二进制工具集

本课题需要使用 binutils 工具集来链接，编译与调试相关编译产物。本课题选用 binutils-riscv64-unknown-elf 来支持面向裸机开发的操作系统程序。binutils 中包含了一系列工具，其中最常用的有 as 汇编器、ld 链接器、objdump 反汇编器、objcopy 目标文件复制器。

(3) 硬件设备的模拟环境

QEMU 是一个开源的硬件模拟器与虚拟机平台，可以支持多种硬件平台的模拟与虚

拟。在本课题的操作系统内核开发中，将使用 QEMU 平台来模拟 RISC-V 架构的硬件平台。

(4) GDB 调试环境

GDB 是一个常用的程序调试工具，能够在程序运行时来实时监控与调试程序，GDB 与 QEMU 通过 Socket 建立 TCP 链接，即可实现本课题操作系统开发的运行时调试工作。

3.2 编译与链接

操作系统的开发中，需要清晰地掌握 C 语言代码和汇编代码到生成最终可执行文件的具体流程，下面将详细说明 C 语言编译到最终的可执行文件的编译流程，并介绍 C 语言如何和汇编语言混合编程及其原理。

(1) 预处理阶段

预处理器是 C 语言编译器中的一部分。它将扫描最初用户编写的源代码文件并且展开所有的宏定义，包含所有的头文件，使用宏处理器指令可实现条件编译来裁剪代码等等。在 GCC 编译器中使用 -E 标志可单独使用其与处理器模块，用于得到宏展开后的 C 语言代码文件。

(2) 编译阶段

编译器将读取预处理阶段产生的预处理后的源代码，并将其翻译为等价的汇编语言指令，汇编语言指令是对机器指令的一种一一映射，仅用于对人类具有良好的可读性，对于不同的 CPU 平台架构，汇编语言指令也有所不同。在 GCC 编译器中使用 -S 标志可单独使用其编译器模块，用于得到编译后的汇编代码。

(3) 汇编阶段

汇编器将读取编译阶段由编译器产生的汇编代码，将其翻译为一一对应的机器码指令，并输出到目标文件，这种机器码是能够被 CPU 直接执行的二进制指令序列。在 GCC 编译器中使用 -c 标志可单独使用其汇编器，用于得到汇编后的目标代码。实际上，GCC 中的汇编阶段只是调用 binutils 工具集中的 as 汇编器的一个便捷入口，开发者也可以直接使用 as 汇编器完成汇编阶段的工作。

(4) 链接阶段

链接器将组合多个目标文件和一些必要的静态链接库文件链接生成一个能够在操作系统平台或裸机上直接运行的单一的可执行文件。在 GCC 编译器中可不加任何标志便能够直接识别到目标文件，将能够使多个目标文件链接为最终的可执行文件。实际上，GCC 中的链接阶段只是调用了 binutils 工具集中的 ld 链接器的一个便捷入口，开发者也可以直接使用 ld 链接器完成链接阶段的工作。

在一个多文件的 C 语言项目中，实际上每个 c 文件都作为一个编译单元依次经过预处理，编译，汇编阶段生成各自的.o 后缀格式的目标文件，最终将所有的.o 后缀格式的目标文件链接为一个最终的可执行文件。当需要和汇编语言进行混合编程时，汇编语言实际上可以直接进入汇编阶段生产出二进制格式的包含机器代码的目标文件。当需要和

其他高级语言混合编程时，也是由其对应编程语言的工具链编译得到目标文件。最终这些目标文件将进入链接阶段，链接阶段不再区分具体的语言，所有的语言均编译成为了目标代码，通过链接器将其链接起来从而实现相互调用。

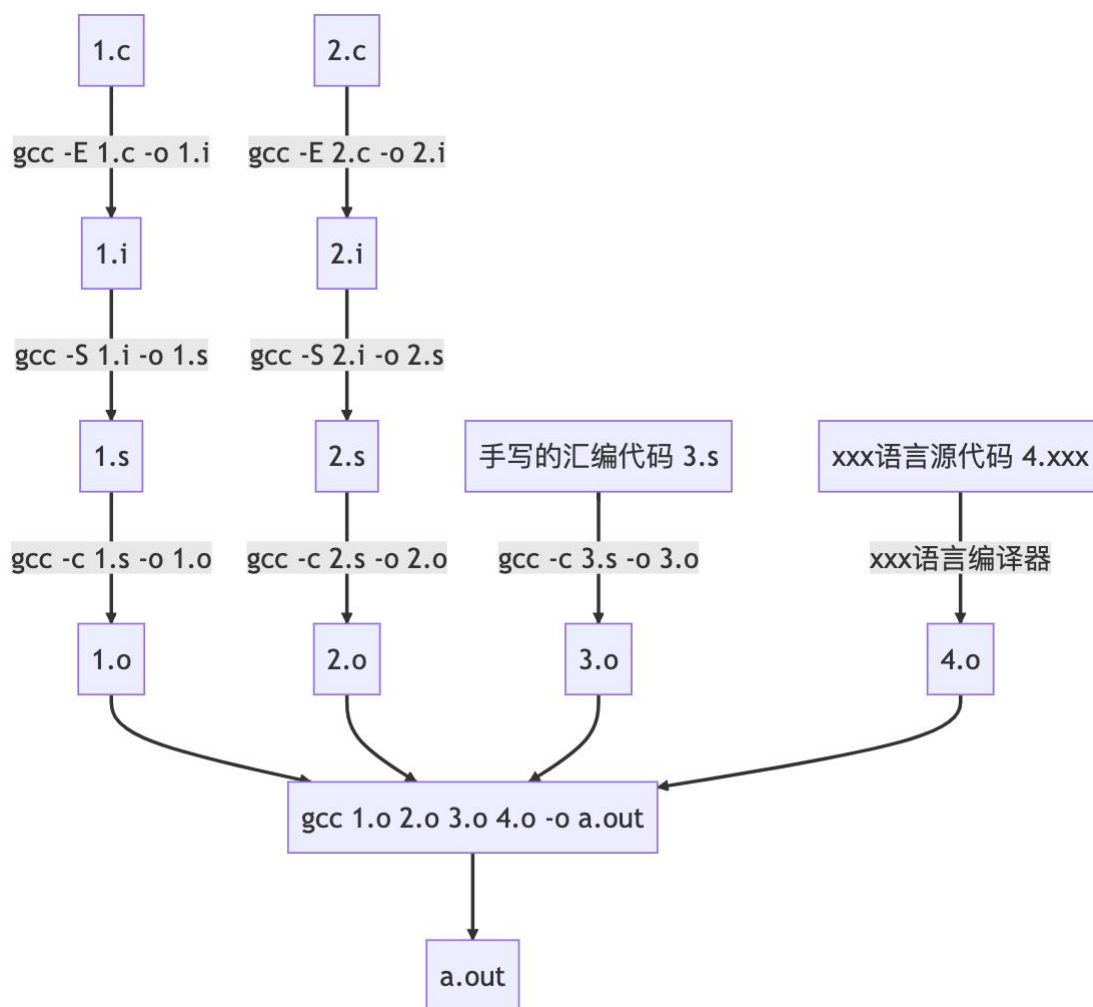


图 3.2 编译流程

如图 3.2 为一个多语言混合项目的完整编译流程，包含 1.c, 2.c 两个 C 语言代码文件经过若干编译流程编译为两个目标文件 1.o, 2.o，包含了开发者自己编写的 3.s 汇编代码通过汇编器汇编为 3.o，还包含了其他语言源代码编译到目标文件 4.o，最终通过链接器将所有目标文件链接为一个可执行文件 a.out。

3.3 从汇编到 C 的引导

实际上，即使是 C 语言这种已经相当贴近硬件底层的语言，最开始时也无法直接编译到裸机运行，而是需要汇编语言这种能够直接操作硬件的语言来为 C 语言准备好运行环境，然后跳转到 C 语言的第一条函数，开始进入操作系统内核执行代码。C 语言的最

核心，最基本的功能便是函数调用，一个可执行的 C 语言程序必然存在至少一个函数调用入口，在常规的 C 语言程序中，这个函数调用入口通常是 `main` 函数，且这个函数还会拥有 `char **argv`, `int argc` 两个形参变量。实际上在 Linux 中，一个 c 语言程序最开始的程序入口是 `_start`，该函数是 `glibc` 系统库的一部分，`glibc` 库最终会经过链接器和用户编写的 c 程序链接到一起形成最终可执行文件可由操作系统调用执行。`_start` 函数将负责为 `main` 函数准备 c 语言函数调用栈的指针 `sp`，初始化静态变量与全局变量，执行全局构造器，准备 `argv`、`argc` 两个参数，跳转到 `main` 函数调用。在本课题实现的操作系统中，也需要实现完成运行内核所需的栈空间和栈指针的初始化。

RISC-V 定义了一个叫做 Hart 的概念，即 hardware thread 硬件线程，多核 CPU 中，多个 CPU 核心可共享同一个物理内存区域，可同时执行同一份代码。传统的多核 CPU 中，每个物理核心在某一时刻仅对应这一个软件线程，随着超线程技术的发展，一个物理核心已经可以同时处理超过一个的软件线程了，即在软件中面对的逻辑上的核心数可以超过实际的物理核心数。在 RISC-V 中，使用了 Hart 术语规范了这些概念，在软件层面不再考虑物理核心数，而只关注逻辑核心数，即 Hart 数。

模拟环境下共有 8 个 hart 数，每个 hart 都拥有一个 id，从 0 开始编号，模拟环境的硬件启动时，所有的 hart 都将同时去加载 `0x80000000` 地址处的代码，即汇编代码中的 `_start` 段，通过读取 `mhartid` 可得到当前执行的 hart 的 id，本课题出于教学操作系统考虑，需要尽可能的简洁易懂，故仅考虑单 hart 的编程，让编号非 0 的 hart 跳转到一种休眠的阻塞状态。让 0 号 hart 继续执行后续代码，设置栈指针，跳转到 C 语言的内核函数的入口。后续便进入 C 语言的入口函数使用 C 语言编写相关的内核初始化代码。

引导程序的启动流程如图 3.3 所示，最终 0 号 hart 进入 C 语言的代码执行流，其他 hart 进入低功耗模式进行空闲等待。

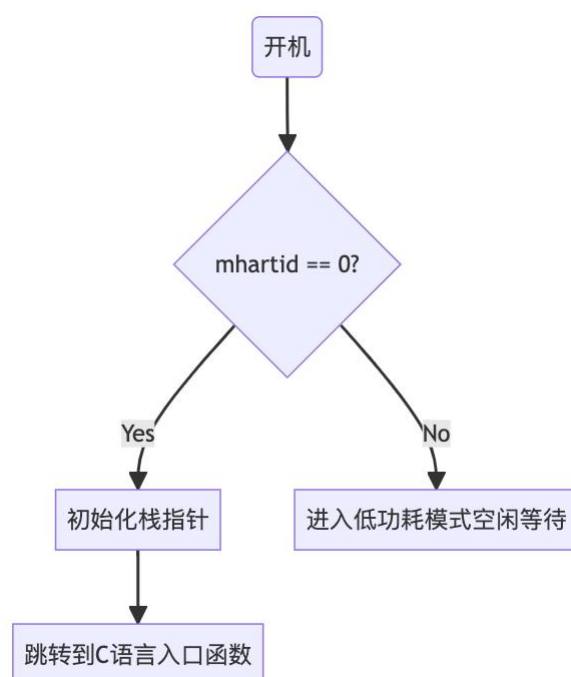


图 3.3 引导程序流程图

3.4 串口设备初始化

本课题所完成的操作系统仅完成一个基本的可供教学演示的内核，与用户的输入输出接口仅考虑命令行即可。在嵌入式开发领域中，常见的与用户交互的输入输出手段便是 UART 接口。

在本课题中，使用 qemu 的 virtio 作为模拟的 SoC 硬件平台，它模拟的串口设备芯片为 NS16550a 芯片，本课题将需要为它编写驱动程序。编写一个设备的驱动程序通常是对该设备提供的若干个寄存器进行读写，经过一层层地封装抽象，最终为上层应用程序开发人员提供了友好的编程接口。在 qemu 模拟的 risc-v 架构的 virtio 硬件下，NS16550a 芯片寄存器被映射为 CPU 地址空间 0x10000000-0x10000100。对这个地址空间下的某些地址单元进行读写操作便映射为了 NS16550a 芯片的寄存器的读写操作，按照 NS16550a 芯片规定的协议指令手册，便可以编写出一个简易的 UART 串口驱动代码。

(1) 串口初始化

初始化一个 NS16550a 串口设备需要完成经过一些初始化步骤，编写相关初始化代码后，便可开始实现串口输出函数。

```
uart_write_reg(IER, 0x00); // 禁用 uart 芯片所有中断信号
```

```
// 设置 LCR 寄存器第 7 位为 1 才能够进入设置 DL 寄存器状态
```

```
uart_write_reg(LCR, uart_read_reg(LCR) | (1 << 7));
```

```
// DL 寄存器低位 0x03, DL 寄存器高位 0x00
```

```
uart_write_reg(DLL, 0x03); uart_write_reg(DLM, 0x00);
```

```
// bit2 停止位 = 0, bit3 不使用奇偶校验
```

```
// bit4 奇校验为 1 偶校验为 0, bit7 启用 DL 寄存器
```

```
uart_write_reg(LCR, 0b00000011);
```

```
// 启用串口接收中断
```

```
uart_write_reg(IER, uart_read_reg(IER) | (1 << 0));
```

(2) 串口输出

向串口输出一个字节本质上就是向串口芯片的 THR 寄存器写入一个字节,但是在写入之前,首先需要检查 THR 寄存器是否正忙,如果正忙则原地自旋等待 THR 寄存器空闲。检查 THR 空闲状态实际上是读取 LSR 寄存器的第 5 位是否为 1 即可实现,具体流程图如图 3.4 代码实现如下。



图 3.4 串口写入流程图

```
// 等待 LSR 寄存器 bit5 为 1 则表示 THR 寄存器空闲
```

```
while ((uart_read_reg(LSR) & LSR_TX_IDLE) == 0);
```

```
return uart_write_reg(THR, ch);
```

3.5 动态内存管理

内存分为了栈空间和堆空间，内核栈空间主要在内核函数执行之前的汇编程序中完成固定大小的初始化，栈空间的使用由编译器编译时自动地完成栈帧的后进先出的管理，除了这部分初始化的汇编代码外，之后无需再手动干涉栈内存空间。

堆内存空间是在运行时动态地由用户手动地开辟和释放的内存空间。本课题将粗略地基于首次最适分配策略完成一个 Page 级别的粗粒度的内存分配算法。

链接器在链接时，将确定处最终程序执行时的内存布局，内存布局信息在 elf 文件中可以确定。通过编写链接器脚本 os.ld，将在链接阶段动态计算出内核代码的各个段内存空间，并将最后一段的所有内存区域划分为堆区。

动态内存管理模块将定义如下接口：

表 3.1 动态内存管理

接口定义	接口描述
<code>void heap_init(void);</code>	初始化页级内存管理模块
<code>void *heap_page_alloc(int npages);</code>	从堆内存区分配 n 个页面大小的内存
<code>void heap_page_free(void *p);</code>	释放堆内存中被分配的页面
<code>void *heap_malloc(size_t __size);</code>	按照给定的字节大小分配内存
<code>void heap_free(void *p);</code>	释放给定指针指向的已分配内存区域

(1) 数据结构定义与初始化

本课题将基于 QEMU 平台作为操作系统运行的模拟平台，它默认能够提供 128MB 的堆内存区域，本课题完成的动态内存分配器将管理这 128MB 的内存区域。为了更高效地管理内存，通常的做法是对内存进行分页管理，通常划分的单个页面大小为 4KB。对于 128MB 的内存区域，需要划分为 $128\text{MB} / 4\text{KB} = 32768$ 个页面大小。

实际上，这 32768 个页面大小对用户而言并不能完全使用，操作系统需要划分出部分内存区域来标记各个内存页面的状态，即元数据信息。对于一个页面而言，至少需要 2 bit 完成页面元数据的记录，即记录页面是否已分配，以及当前页是否属于已分配内存区域的末尾标志。本课题对每个页面使用了一个字节即 8bit 作为页面的元数据记录。那么，总计有 32768 个页面，应当需要划分多少个页面区域作为元数据区域呢？

可以假设 32768 个页面中有 x 个用户可使用页面，则需要 x 字节存放元数据，可列出不等式如下：

$$x + \frac{x}{4096} \leq 32768$$

解得

$$x \leq 32760.0019$$

考虑到实际意义, x 为页面数, 故 x 为整数, 得

$$x = 32760$$

故用户可用 32760 个内存页面, 需要 8 个内存页面作元数据信息的存放区域。

在了解了数据结构的定义后, 便可编写堆内存区域的初始化操作, 实现代码如下:

```
struct Page { uint8_t flags; };
void heap_page_init() {
    pages_nums = 32760; // 总页面数
    // 初始化_num_pages 个页描述符, 128M 内存 _num_pages = 32760
    for (struct Page *page = (struct Page *)HEAP_START;
        page < (struct Page *)HEAP_START + page_nums;
        page++) { _clear(page); }
    // 向后移动 8 块内存页面作为堆区的起始地址
    heap_start = HEAP_START + 8 * PAGE_SIZE;
    // 计算堆区的结束地址
    heap_end = _alloc_start + (PAGE_SIZE * page_nums);
}
```

(2) 页级内存分配

```
void *heap_page_alloc(int npages) {
    int found = 0;
    struct Page *page_i = (struct Page *)HEAP_START;
    // 滑动窗口搜索
    for (int i = 0; i <= (_num_pages - npages); i++) {
        // 非空闲区域, 则继续向后寻找
        if (!_is_free(page_i)) {
            page_i++;
            continue;
        }
        // 此时找到了第一个空闲页面
        found = 1;

        // 继续向后搜寻, 若后面的 npages - 1 个页均为未分配区域, 那么区域搜索成功
        struct Page *page_j = page_i + 1;
        for (int j = 1; j < npages; j++) {
            // 搜索过程发现了一个已分配区域, 跳出此次循环, 搜索失败
            if (!_is_free(page_j)) {
                found = 0;
                break;
            }
            page_j++;
        }
    }
}
```

图 3.5 堆区页级内存分配

如图 3.5 函数 `heap_page_alloc` 是基于首次最适内存分配策略实现的用于在堆空间中分配连续的内存页的页级内存分配算法, 实际上具体代码实现是一个基于双指针的搜索算法。传入参数 `npages` 表示需要分配的连续内存页数。函数首先从堆的起始地址开始, 通过双指针来逐步动态扩大滑动窗口的搜索方式来查找空闲的内存页区域。若搜索到一

个空闲的内存页，则继续向后查找，不断检查是否有足够的连续内存页可供分配。如果可用内存页数满足要求，则标记这些内存页为已分配状态，并返回这些内存页的起始地址。如果搜索不到可用的内存页，则返回 NULL 表示分配失败。

(3) 页级内存回收

```
void heap_page_free(void *p) {
    // 内存分配失败后返回0地址，用户可能会将0地址进行错误地释放
    // 这里进行一下判定并增加越界检查
    if (!p || (uint32_t)p >= _alloc_end) return;

    // 第一个页描述符
    struct Page *page = (struct Page *)HEAP_START;

    // 计算当前页面的编号并找到当前页面的页描述符指针
    page += ((uint32_t)p - _alloc_start) / PAGE_SIZE;

    while (!_is_free(page)) {
        _clear(page);
        if (_is_last(page)) break;
        page++;
    }
}
```

图 3.6 堆区页级内存回收

如图 3.6 函数 heap_page_free 实现了堆空间中分配的连续内存页的释放算法。函数传入参数 p 为需要释放的内存页的起始地址。函数首先对传入参数进行判断，如果 p 为 NULL 或者超出堆空间的范围，则直接返回，避免出现非法的内存页释放。如果传入参数合法，则通过计算当前页面的编号并找到当前页面的页描述符指针。然后，函数将该内存页的页描述符标记为未分配状态，并检查该内存页是否为最后一页。如果不是，则继续向后查找并标记下一个内存页的页描述符。如果找到了最后一页，则直接退出循环。最终，该函数完成了堆空间中一个已分配的连续内存页区域的释放操作。

(4) malloc/free 的简单实现

```
void *malloc(size_t __size) {
    int need_use_pages = __size / PAGE_SIZE;
    if (__size % PAGE_SIZE > 0) {
        need_use_pages++;
    }
    return heap_page_alloc(need_use_pages);
}

void free(void *p) { heap_page_free(p); }
```

图 3.7 malloc 与 free 实现代码

如图 3.7 这两个函数是 C 语言中常用的内存分配和释放函数。`malloc` 函数用于在堆空间中分配指定大小的内存，传入参数 `_size` 表示需要分配的内存大小。该函数首先计算需要使用的内存页数，然后调用 `heap_page_alloc` 函数来进行页内存分配，将分配得到的内存的起始地址返回给调用者。

`free` 函数用于释放指定的内存，传入参数 `p` 为需要释放的内存的起始地址。该函数释放内存的原理是将该内存页的页描述符标记为未分配状态，以便下次 `malloc` 函数调用时可以重用这部分内存。该函数具体的实现是通过调用 `heap_page_free` 函数来完成内存释放操作。

3.6 进程管理与调度

嵌入式操作系统中，任务是被调度的最小单位，在 C 语言中通常是一个死循环的函数，但是这些函数的执行过程却互不阻塞。实际上嵌入式操作系统中的任务就对应于通用操作系统中的进程、线程的概念。进程是操作系统进行资源调度和分配的最小单位，线程是操作系统调度的最小单位。嵌入式操作系统中，用户程序最终将与操作系统内核程序链接在一起形成一个完成的程序，故一个嵌入式的用户程序通常类似于一个单进程多线程的传统操作系统上的程序。通用操作系统中，进程控制块 PCB (Process Control Block)，线程控制块 TCB (Thread Control Block) 是操作系统感知进程存在与线程存在的唯一途径，在本课题完成的嵌入式操作系统中，使用任务控制块 TCB (Task Control Block) 来唯一确定表示一个任务，并包含了每个任务自身所需的全部资源字段，如上下文寄存器，任务栈，任务状态，任务参数等。为了和教科书中的术语保持一致，不妨约定后续的 PCB 均可指代这里的 TCB，进程均可指代这里的任务。

本课题计划需要完成如下任务管理相关的操作接口：

表 3.2 多任务管理与调度接口

接口定义	接口描述
<code>void task_create(void (*entry)(), void *params)</code>	创建一个任务 entry: 传递一个函数指针指定一个任务的执行入口 params: 传递一个参数指针来指定该任务的自定义参数数据
<code>void task_yield()</code>	让出当前任务的 CPU 使用权, 使当前进程进入就绪态
<code>void *task_get_current_params()</code>	获取当前任务的参数, 返回一个用户自定义的任意类型数据的指针
<code>int task_get_current_id()</code>	获取当前任务的 id
<code>void task_block()</code>	使当前任务进入阻塞态
<code>void task_wakeup(int id)</code>	根据某个任务 id 来唤醒某个任务, 使之进入就绪态, 等待任务调度器下一次调度使之进入运行态。
<code>void task_destroy()</code>	销毁当前正在执行的任务
<code>void schedule()</code>	按某种调度策略调度选取一个任务上 CPU 运行并使之进入运行态

(1) 进程控制块

代码中的结构体 task 便是 TCB, 其中包含了任务 id, 任务参数, 任务栈, 任务上下文, 任务状态字段。MAX_TASKS 常量表示 TCB 数组的大小, 即所能创建的最大任务数目。STACK_SIZE 常量表示每个任务的任务栈的最大长度, C 语言函数调用, 局部变量等均会使用任务栈, 若某个任务的函数调用层级过深, 可能会造成任务栈溢出, 此时可考虑增加任务栈的大小。全局变量_size 表示当前被创建的总任务数, 每次调用 task_create()成功创建任务之后该值应当加一, 当调用 task_destroy()成功销毁任务之后该值应当减一。全局变量_current 表示当前处于运行态的任务指针, 由于本课题的操作系统仅考虑单核 CPU 上的任务调度, 故同一时刻仅存在一个运行态的任务。

```
// 每个进程的最大栈空间数
#define STACK_SIZE 2048
// 最大进程数
#define MAX_TASKS 10
// 定义进程状态
enum task_state { READY, RUNNING, BLOCKED, END };
// 定义 PCB 数组
static struct task {
```

```

// 任务 id
uint8_t id;
// 任务参数
void params;
// 任务栈
uint8_t stack[STACK_SIZE];
// 任务上下文
struct context context;
// 任务状态
enum task_state state;
} _tasks[MAX_TASKS];
static int _size = 0, _current = -1;
(2) 进程状态转换

```

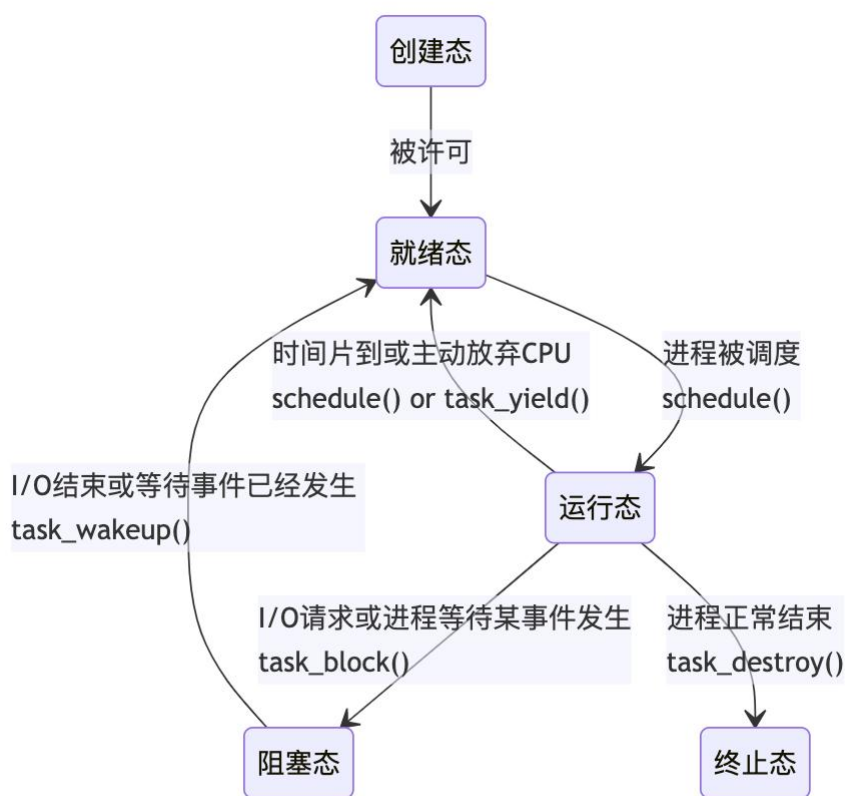


图 3.8 进程状态转换图

在操作系统理论课程中，进程状态转换是一个重要的知识点，其转换过程如图 3.8，本课题完成的操作系统中，进程状态转换也是一个相当重要的机制，基于进程状态转换，实现了进程阻塞状态下的让权等待，而不会阻塞状态下依然占据 CPU 的使用权。本课题在 PCB 中定义了枚举类型 `task_enum` 的字段，标识当前进程的状态。当进程被创建后，在本课题中实现的操作系统中实际上将跳过创建态直接进入了就绪态等待 CPU 随时进

行任务调度，当被进程调度器成功调度后，进程将进入运行态，进程将开始运行，进程在运行状态可以随时调用 `task_block()`, `task_yield()`, `task_destroy()` 这三个进程来改变自身状态选择进入阻塞态、回到就绪态或终止当前进程运行。

(3) 时间片轮转调度算法

`schedule()` 函数便为任务调度的入口，如图 3.9 为该函数的程序流程图，操作系统将通过定时器中断按固定的时间间隔反复调用 `schedule()` 函数，`schedule()` 函数负责按照某种策略去选取一个任务上 CPU 运行，本课题完成了一个相当简单的时间片轮转调度算法 (Round-Robin) 的实现。当操作系统调用 `schedule()` 函数时，将发生进程的切换，首先设置当前正在执行的任务为就绪态，通过轮转寻找到下一个处于就绪态的任务，设置该任务为运行态，切换到该任务的上下文，之后处理器将在 `switch_to()` 函数中，按照更新后的上下文中的 PC 寄存器继续完成切换后任务的执行流程。特别地，若当前进程被调用了 `task_block()` 内核函数，则当前进程状态将被设置为阻塞状态，并且 `task_block()` 函数内部直接以内部软中断的方式触发 `schedule()` 函数进行让出 CPU 使用权，触发进程调度，此时在当前进程未被唤醒前，`schedule()` 函数将总是跳过该被阻塞的进程，使进入阻塞态的进程无法得到上处理器运行的机会，这就是基于进程的状态转换来实现进程阻塞的原理。实际上，处理器是一个有状态的硬件，其中各个寄存器的值便为处理器的状态数据，统称为上下文，代码中 `switch_to` 函数的实现中 `context.S` 中，是一段简单的汇编代码来完成 CPU 的上下文的切换。

```
void schedule() {
    // 不存在待调度的任务
    if (_size <= 0) return;
    // 令当前正在调度的任务回到就绪态
    if (_tasks[_current].state == RUNNING) {
        _tasks[_current].state = READY;
    }
    // 寻找下一个就绪态的任务
    do {
        _current = (_current + 1) % _size;
    } while (_tasks[_current].state != READY);
    // 切换到运行态
    _tasks[_current].state = RUNNING;
    // 切换 CPU 上下文
    switch_to(&_tasks[_current].context);
}
```

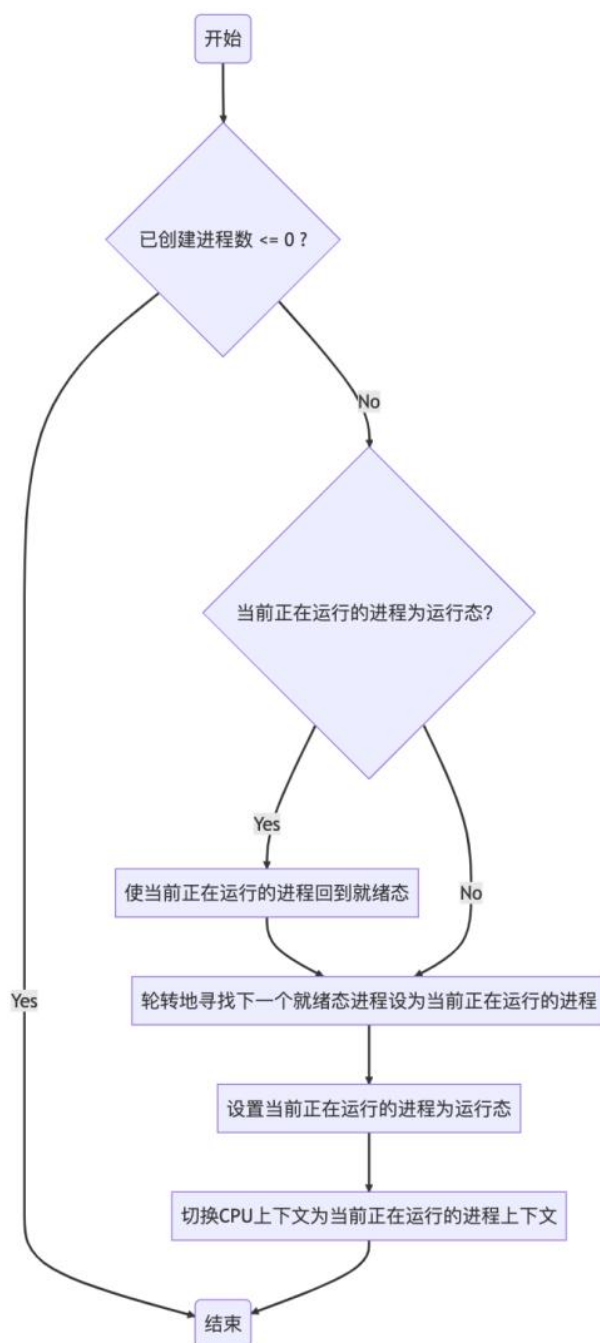


图 3.9 进程调度器流程图

3.7 进程同步

3.7.1 自旋锁

本课题计划将基于 CAS 机制实现一个自旋锁，可保证某个资源的互斥访问，即使某任务的临界区未执行完成而被进程调度器调度到其他进程，其他进程也无法进入临界区，这也保证该临界区代码执行的原子性。

实现方法可以编写一个 spinlock 结构体，spinlock 结构体中存放了锁的所有信息，

本课题为简单期间，仅使用一个 `int` 类型的 `locked` 变量标记锁状态，该变量为二值化的，取值仅为 0 或 1。`locked` 为 0 表示未上锁，可直接获得锁，`locked` 为 1 表示已上锁，其他任务获得锁需要原地自旋等待。

自旋锁模块提供的接口如下：

表 3.3 自旋锁接口定义

接口定义	接口描述
<code>void lock_init(struct spinlock *)</code>	初始化一个锁
<code>void lock_acquire(struct spinlock *)</code>	获得一个锁
<code>void lock_release(struct spinlock *)</code>	释放一个锁

(1) 获得锁

```
void lock_acquire(struct spinlock *lock) {
    bool success;
    do {
        if(lock -> locked == 0) { // Compare 比较
            lk -> locked = 1; // Set 设置
            success = true; // 成功则退出循环，否则将继续
        }
    } while(!success);
}
```

上述代码中，当 `locked` 为 0 时表示不存在临界区代码正在执行，需要进入临界区的任务可直接获得锁进入临界区执行临界区代码，获得锁之后，需要置 `locked` 变量为 1，其他任务进入临界区之前获得锁将原地循环查询变量是否为 0，如果非 0，则继续原地循环。

这种不断循环比较 `locked` 是否等于 0 并且设置 `locked` 等于 1 的方法叫做 CAS (Compare And Set) 机制。但是上述的获得锁的实现代码并不是原子性的，若编译为汇编代码，可发现上述获得锁中 CAS 的代码由 `bnez` (branch not equal zero) 指令实现分支跳转，由 `li` 指令实现置 `locked` 为 1。在并发场景下，将可能会导致任务 A 获得锁后准备执行 `locked = 1` 指令，此时被上下文切换到任务 B，任务 B 仍然能够获得锁并执行 `locked = 1` 上锁，此时任务 A 与任务 B 将同时进入临界区，将会会造成并发安全问题。

为了解决这个问题，CPU 在硬件层面专门提供了 CAS 原子指令 `amoswap`，仅需使用一条指令即可完成上述代码的 CAS 动作，该硬件指令提供了原子 CAS，修改后的代码如下：

```
void lock_acquire(struct spinlock *lock) {
    bool success;
    do {
        success = __sync_lock_test_and_set(&lk->locked, 1);
    }while(!success);
}
```

```
}
```

(2) 释放锁

```
void lock_release(struct spinlock *lock) {  
    lock -> locked = 0;  
}
```

上述代码实现了释放锁的逻辑，仅需直接令锁中的 `locked` 标记变量置为 0，当临界区代码执行完毕后需要执行 `release` 操作来释放锁。此时将置 `locked` 变量为 0，若其他任务在某一时刻轮询发现 `locked` 变量为 0，则将能够直接获得该锁，此时将结束循环进入临界区。

3.7.2 信号量

信号量 (Semaphore) 是一种更加强大的进程同步机制，它用于在多进程环境下实现对有限资源的访问控制，本质上是一种计数器，来表示某个资源的可用数量，它可以实现在多进程下的资源合理竞争和避免死锁等并发相关的问题。

本课题将完整地以教科书式的方式复现出信号量 PV 操作原语的实现过程。

表 3.4 信号量接口定义

接口定义	接口描述
<code>void semaphore_init(semaphore *s, int n)</code>	信号量的初始化
<code>void semaphore_wait(semaphore *s)</code>	信号量的 wait 操作
<code>void semaphore_signal(semaphore *s)</code>	信号量的 signal 操作

(1) 数据结构定义

传统的操作系统教科书中，定义的信号量的数据结构如下：

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

每个信号量都是一个结构体对象，它包含了一个整数 `value` 和一个进程链表 `list`，当一个进程在该信号量上使用 `wait` 操作发生阻塞等待时，将该进程的 PCB 加入该链表，当使用 `signal` 操作后，将从 `list` 进程链表中取走一个进程的 PCB 并唤醒该进程。通常最先取走的进程 PCB 为最先加入的进程 PCB，即先进先出，故该链表 `list` 又称为等待队列。

在本课题实现的信号量模块中，定义的信号量和等待队列的数据结构如下：

```
// 等待队列的最大长度为 10  
#define SEMAPHORE_MAX_VALUE 10  
  
// 信号量  
typedef struct {  
    int value;          // 当前信号量的值  
    SqQueue queue;     // 等待队列
```



```

    struct spinlock wait_lock;    // wait 操作锁
    struct spinlock signal_lock;  // signal 操作锁
} semaphore;

```

本课题使用了环形队列来保存各个进程的 pid 来实现 list 等待队列, 并且添加额外的两个 spinlock 来保证 wait 操作和 signal 操作的原子性。

(2) init 操作

传统操作系统教科书中通常不定义信号量的初始化操作, 伪代码中初始化信号量和直接一个普通 int 变量的赋值相同。

本课题中 init 操作实现如下:

```

// 初始化信号量
void semaphore_init(semaphore *s, int value) {
    s->value = value;
    InitSqQueue(&s->queue);
}

```

(3) wait 操作

传统操作系统教科书中定义的信号量的 wait 操作如下, 且通常还应当强调该 wait 操作的实现必须保证为原子的:

```

void wait(semaphore *s) {
    s -> value--;
    if (s -> value < 0) {
        将当前进程加入 s->list
        block();    // 阻塞当前进程
    }
}

```

本课题完成信号量的 wait 操作代码如下, 使用了自旋锁的 acquire 和 release 操作, 来保证 wait 操作的原子性, 使用了环形队列来代表等待队列, 保存等待该信号量的进程 pid。当一个进程执行 wait 操作的 value--后信号量的值为负数, 则说明资源不足, 为保证忙则等待的原则, 进程将进入等待状态。为保证让权等待, 应当使不能立即进入临界区的进程释放 CPU 执行权, 操作系统需要将当前运行的进程放入等待队列, 并调用 task_block()函数设置当前进程为阻塞态并让出 CPU 执行权, 调度器将调度下一个处于就绪态的进程。

```

void semaphore_wait(semaphore *s) {
    lock_acquire(&s->wait_lock);
    s->value--;
    if (s->value < 0) {
        EnQueue(&s->queue, task_get_current_id());
        task_block();
    }
    lock_release(&s->wait_lock);
}

```

(4) signal 操作

传统操作系统教科书中定义的信号量的 signal 操作如下,且通常还应当强调该 signal 操作的实现必须为原子的:

```
void signal(semaphore *s) {
    s->value++;
    if (s -> value <= 0) {
        选择 s->list 中的进程 P 并删除
        wakeup(P); // 唤醒进程 P
    }
}
```

本课题完成的信号量 signal 操作代码如下,使用了自旋锁的 acquire 和 release 操作,来保证 signal 操作的原子性,当信号量执行完 value++操作后,发现 value 的值仍然非正数,则说明正在有进程处于等待资源的阻塞状态,需要由操作系统唤醒一个阻塞状态的进程。为了确保有限等待的准则,可使用 FIFO 策略,即按照先进先出的策略取出所有等待该信号量的进程中最先进入阻塞状态的进程 pid,并按照该 pid 调用 task_wakeup(id)操作来唤醒该阻塞进程,使该进程进入就绪态,等待被操作系统随时调度恢复运行。

```
void semaphore_signal(semaphore *s) {
    lock_acquire(&s->signal_lock);
    s->value++;
    if (s->value <= 0) {
        int id;
        DeQueue(&s->queue, &id);
        task_wakeup(id);
    }
    lock_release(&s->signal_lock);
}
```

3.8 文件系统

实现一个文件系统通常应当包含以下操作:

- (1) 创建文件: 创建一个新的文件, 并为其分配磁盘空间。
- (2) 删除文件: 删除一个已经存在的文件, 并释放其占用的磁盘空间。
- (3) 读取文件: 从文件中读取数据, 并将其返回给调用者。
- (4) 写入文件: 将数据写入文件中, 并将其保存到磁盘上。
- (5) 打开文件: 打开一个已经存在的文件, 并返回一个文件句柄, 以便后续的文件操作可以使用该句柄。
- (6) 关闭文件: 关闭一个已经打开的文件句柄, 并释放相关的资源。
- (7) 移动文件指针: 将文件指针移动到文件中的指定位置, 以便后续的读写操作可以从

该位置开始。

- (8) 获取文件信息：获取文件的元数据信息，如文件的大小、创建时间、修改时间等。
- (9) 修改文件信息：修改文件的元数据信息，如文件的访问权限、修改时间等。
- (10) 创建目录：创建一个新的目录，并为其分配磁盘空间。
- (11) 删除目录：删除一个已经存在的目录，并释放其占用的磁盘空间。
- (12) 遍历目录：遍历一个目录中的所有文件和子目录，并返回它们的名称和元数据信息。

3.8.1 FatFs 项目概述与移植

本课题将尝试移植 FatFs 项目，为了简化复杂性，将直接读写内存中模拟一个磁盘设备并基于这种内存中模拟的磁盘设备移植 FatFs 项目，最终完成 FAT32 文件系统的格式化与各种文件系统相关的操作。

FatFs 是专用于小型嵌入式操作系统的通用 FAT/exFAT 文件系统的模块，它与具体的磁盘 I/O 完全隔离，独立于具体的平台，易于移植到自己的嵌入式项目，支持多种配置与工作模式。根据 FatFs 项目官方文档，FatFs 项目对用户提供了如下的 API 接口函数供用户使用，为了方便起见，不妨省略函数接口的参数类型和返回类型：

文件访问相关接口如表 3.5 文件访问相关接口

表 3.5 文件访问相关接口

接口定义	接口描述
f_open(fp, path, mode)	打开或创建一个文件
f_close(fp)	关闭一个已经打开的文件
f_read(fp, buff, btr, br)	从指定文件读取数据
f_write(fp, buff, btw, bw)	写入数据到指定文件
f_lseek(fp, ofs)	移动读写指针，扩展文件大小
f_truncate(fp)	扩充文件大小到当前的文件读写指针
f_sync(fp)	刷新缓存数据
f_forward(fp, func, btf, bf)	读取并转发数据到流
f_expand(fp, fsz, opt)	分配一个连续的块到文件中
f_gets(buff, len, fp)	读取文件为一个字符串
f_putc(ch, fp)	写入一个字符到文件
f_puts(str, fp)	写入一个字符串到文件
f_printf(fp, fmt, ...)	写入一个格式化的字符串到文件
f_tell(fp)	获取当前文件的读写指针
f_eof(fp)	探测文件是否结束
f_size(fp)	获取文件大小
f_error(fp)	测试文件是否有误

目录访问相关接口如表 3.6 目录访问相关接口

表 3.6 目录访问相关接口

接口定义	接口描述
f_opendir(dp, path)	打开一个目录
f_closedir(dp)	关闭一个已打开的目录
f_readdir(dp, fno)	读取目录项
f_findfirst(dp, fno, path, pattern)	打开一个目录并读取第一个匹配项
f_findnext(dp, fno)	读取下一个匹配项

文件和目录管理相关接口如表 3.7 文件与目录管理相关接口

表 3.7 文件与目录管理相关接口

接口定义	接口描述
f_stat(path, fno)	检查文件或子目录是否存在
f_unlink(path)	删除文件或子目录
f_rename(old_name, new_name)	重命名/移动文件或子目录
f_chmod(path, attr, mask)	更改文件或子目录属性
f_utime(path, fno)	更改文件或子目录的时间戳
f_mkdir(path)	创建一个子目录
f_chdir(path)	改变当前工作目录
f_chdrive(path)	改变当前驱动器
f_getcwd(buff, len)	获取当前的工作目录和驱动器

卷管理与系统配置相关接口如表 3.8 卷管理与系统配置相关接口

表 3.8 卷管理与系统配置相关接口

接口定义	接口描述
f_mount(fs, path, opt), f_unmount(path)	挂载/卸载卷
f_mkfs(path, opt, work, len)	在逻辑驱动器上创建 FAT 卷
f_fdisk(pdrv, ptbl, work)	在物理驱动器上创建一个分区
f_getfree(path, nclst, fatfs)	获取卷的可用空间
f_getlabel(path, label, vsn)	获取卷标
f_setlabel(label)	设置卷标
f_setup(cp)	设置活动代码页

由于 FatFs 模块独立于具体的平台与具体的存储介质，它仅仅是数据持久化中的文件系统层，并不负责磁盘 I/O 层的实现，它完全脱离了具体存储的物理设备，为了移植 FatFs，操作系统需要为 FatFs 实现一些磁盘操作相关的底层接口，操作系统需要提供底层磁盘 I/O 接口如表 3.9 磁盘 I/O 相关接口

表 3.9 磁盘 I/O 相关接口

接口定义	接口描述
disk_status(pdrv)	获取磁盘状态

<code>disk_initialize(pdrv)</code>	初始化磁盘设备
<code>disk_read(pdrv, buff, sector, count)</code>	读取磁盘数据
<code>disk_write(pdrv, buff, sector, count)</code>	写入磁盘数据
<code>disk_ioctl(pdrv, cmd, buff)</code>	控制设备特定的功能函数
<code>get_fattime()</code>	获取当前 RTC 时间

本课题为了简化移植工作，使用一块内存来模拟一个磁盘设备，并使用 FatFs 项目针对该内存区域模拟的磁盘设备进行分为两块磁盘分区，并对 0 号分区执行 FAT32 文件系统的格式化操作。

(1) 初始化一段内存作为模拟的磁盘空间

```
// 开辟 10MB 的内存空间作为 ramdisk
#define SECTOR_SIZE 512
#define SECTOR_COUNT 20000
#define RAMDISK_SIZE (SECTOR_SIZE * SECTOR_COUNT)
static BYTE ramdisk[RAMDISK_SIZE];
```

这里模拟的磁盘单个扇区有 512 字节，总计 20000 个扇区，即 $512\text{B} * 20000 = 10\text{MB}$ 大小的空间。

(2) 实现磁盘读取接口 `disk_read()`

```
// 将 ramdisk 中的数据拷贝到 buff 中
// 偏移量为 sector * SECTOR_SIZE
// 拷贝的数据长度为 count*512
BYTE *dst = buff;
const BYTE *src = ramdisk + sector * SECTOR_SIZE;
UINT n = count * SECTOR_SIZE;
while (n--) *dst++ = *src++;
```

将起始于 sector 号，count 数量的连续扇区中的数据逐字节地依次拷贝到 buff 指针起始的空间，即可成功实现磁盘读取接口。

(3) 实现磁盘写入接口 `disk_write()`

```
// 将 buff 中的数据，拷贝到 ramdisk 中
// 偏移量为 sector * SECTOR_SIZE
// 拷贝的数据长度为 count*512
BYTE *dst = ramdisk + sector * SECTOR_SIZE;
const BYTE *src = buff;
UINT n = count * SECTOR_SIZE;
while (n--) *dst++ = *src++;
```

将 buff 指针开始的内容依次复制到起始于 sector 号，count 数量的连续扇区中的空间，即可实现磁盘写入接口。

(4) 实现磁盘控制接口 `disk_ioctl()`

```
switch (cmd) {
    case GET_SECTOR_COUNT: *(DWORD *)buff = SECTOR_COUNT; break;
```

```

        case GET_SECTOR_SIZE: *(DWORD *)buff = SECTOR_SIZE; break;
    }

```

此处必须实现两个磁盘设备的 io 控制指令，分别用于获取扇区大小与获取可用扇区数目，获取到的结果为 DWORD 类型，写入到 buff 指向的内存中。如果缺少这两个 io 控制指令，将可能影响部分上层 API 接口的正常行为，如可能无法正常格式化磁盘，无法完成磁盘分区等。

3.8.2 磁盘分区与格式化操作

计划对 10MB 的 0 号 RAM 磁盘划分两个分区，每个分区为大小均等的 5MB。首先编写分区挂载记录为全局变量如下，它将在链接阶段链接到 FatFs 项目内部。它记录了两个逻辑分区，这两个逻辑分区号分别为数组下标 0, 1。其中逻辑分区 0 映射为 0 号物理磁盘上的 1 号分区，逻辑分区 1 映射为 0 号物理磁盘上的 2 号分区。注意物理磁盘分区的分区号从 1 开始计数。

```

// 定义磁盘逻辑分区映射表，由物理磁盘号，分区号构成的数组
PARTITION VolToPart[FF_VOLUMES] = {{0, 1}, {0, 2}};

```

这一行代码完成磁盘的分区工作，并校验 0 扇区的 MBR 主引导记录是否正确，MBR 共占据一个扇区，即 512 字节，它在最后的两个字节为固定的 0x55, 0xAA，磁盘分区成功后，断言一下 MBR 的尾部两字节是否正确。然后格式化逻辑 0 号分区。

```

BYTE work[FF_MAX_SS]; // 工作缓冲区
LBA_t plist[] = {50, 50, 0}; // 磁盘分为两个各占 50%大小的分区
f_fdisk(pdrv, plist, work); // 磁盘分区
disk_read(pdrv, work, 0, 1); // 读取磁盘的 0 扇区的 512 字节到 work 数组
f_mkfs("0:", 0, work, sizeof(work)); // 格式化 0 号分区

```

3.8.3 文件的创建与读写

这段代码完成一个名为 test.txt 的文本文件创建并写入长度 14 字节的字符串数据，然后关闭文件，获取文件信息并输出该文件的大小，文件属性，文件名称。然后再次打开 test.txt 文件，获取文件大小并创建缓冲区，读取文件内容并输出到缓冲区，输出缓冲区中的字符串内容。

```

FIL fp; FILINFO fno;
f_open(&fp, "test.txt", FA_CREATE_ALWAYS | FA_WRITE); // 创建文件
f_write(&fp, "Hello, world!", 14, NULL); // 写入文件
f_close(&fp); // 关闭文件
f_stat("test.txt", &fno); // 获取文件信息
printf("size: %d\n name: %s\n", fno.fsize, fno.fname); // 文件信息
f_open(&fp, "test.txt", FA_READ); // 打开文件
FSIZE_t sz = f_size(&fp); char buff[sz]; // 获取文件大小并创建缓冲区
f_read(&fp, buff, sizeof(buff), &bytes_read); // 读取文件到缓冲区
printf("File content: %s\n", buff); // 输出已读取长度文件内容
f_close(&fp); // 关闭文件

```

3.8.4 目录创建与目录列举

下面将创建一个目录文件，目录文件为 FAT32 中的一种特殊的文件格式，它具有特殊标记可被 FAT32 文件系统驱动所识别为文件夹，目录文件内容存放了其目录中包含的文件簇号相关数据。以下代码将在根目录创建一个 `test_dir` 文件夹，并遍历根目录。

具体实现代码如下：

```
f_mkdir("test_dir"); // 创建目录
DIR dir; f_opendir(&dir, ""); // 打开目录
FILINFO fno; f_readdir(&dir, &fno); // 读取目录
// 遍历并输出目录中的内容
FRESULT fr = f_findfirst(&dir, &fno, "", "*");
while (fr == FR_OK && fno.fname[0]) {
    printf("find[%s]: %s\n",
        fno.fattrib == AM_DIR ? "DIR" : "FILE", fno.fname
    );
    fr = f_findnext(&dir, &fno);
}
```

4 用户程序案例实现

4.1 链表数据结构

本课题操作系统在内核态实现了一个简易的内存分配器，对用户态程序提供了 malloc() 和 free() 接口，本案例基于内核程序的内存分配器提供的接口实现了一个链表操作的演示程序。由于本课题设计的内存分配器最小粒度以页为单位，单页大小设置为 4KB，故链表中的一个结点占用即使小于 4KB 大小，也至少需要占据 4KB 的内存空间。

(1) 链表结点结构体定义

```
typedef struct Node {  
    int data;  
    struct Node *next;  
} Node;
```

这段代码定义了一个链表结点，它由数据域 data 和指针域 next 所构成，其中数据域假定为 int 类型，gcc 的 int 类型在 RISC-V 的 32 位平台占用大小为 4 字节。指针域指向了下一个结点，在 RISC-V 的 32 位平台上占用大小为 4 字节，在 64 位平台上占用大小为 8 字节，故最终该结构体在 32 位平台占用大小为 8 字节，但由于内存分配器最小分配粒度以页为单位，单页大小为 4KB，故所开辟的内存空间为 1 个页。

(2) 初始化链表

```
Node *createList() {  
    Node *head = (Node *)malloc(sizeof(Node));  
    head->next = NULL;  
    return head;  
}
```

(3) 头插法插入链表元素

```
void insertList(Node *head, int data) {  
    Node *node = (Node *)malloc(sizeof(Node));  
    node->data = data;  
    node->next = head->next;  
    head->next = node;  
}
```

(4) 遍历链表

```
void traverseList(Node *head) {  
    for (Node *p = head; p->next != NULL; p = p->next) {  
        printf("%d ", p->next->data);  
    }  
}
```

(5) 运行结果展示

选择插入数据选项，依次插入 12, 23, 34, 45 数据，输入 q 结束插入，然后选择遍

历数据选项，则完成链表的遍历输出。

```
链表演示程序
0. 插入数据
1. 删除数据
2. 查找数据
3. 遍历数据
4. 销毁链表
5. Exit
Use 'w', 's' to move choice
Use 'q' to exit
Use 'Enter' to run choice
请输入要插入的数据: 12
请输入要插入的数据: 23
请输入要插入的数据: 34
请输入要插入的数据: 45
请输入要插入的数据: q
遍历链表: 45 34 23 12
Press any key to continue
┐
```

图 4.1 运行结果

4.2 生产者—消费者演示程序

生产者消费者问题，又称有限缓冲问题，是操作系统领域中并发同步的经典模型案例。它描述的是存在 n 个进程作为生产者， m 个进程作为消费者，生产者不断生产数据放入缓冲区中，消费者在缓冲区中消费这些数据。在生产者消费者问题中，一个重要的保证就是当缓冲区满后，生产者暂停生产数据，等待缓冲区不满后继续恢复生产，当缓冲区空后，消费者暂停消费数据，等待缓冲区不空后继续恢复消费。信号量机制是处理生产者消费者问题的一个经典有力的工具，它能够较好的处理多生产者多消费者的并发同步问题。本课题的操作系统实现了信号量机制，基于此完成了生产者消费者模型的一个经典代码实现。

(1) 定义缓冲区队列

使用环形队列定义一个缓冲区 `buffer`，缓冲区大小为 3，缓冲区具有 `enqueue` 和 `dequeue` 方法可进行入队和出队操作。

(2) 定义信号量与相关变量

为了模拟生产者和消费者所操作的数据，使用变量 i 作为产品的唯一标识，每生产一资源，变量 i 将自增。为了确保变量 i 不会同时被多个生产者同时自增，使用一个互斥信号量 `mutex_i` 来保护变量 i ，使得变量 i 能够被多个生产者互斥地访问。

生产者消费者模型中的缓冲区使用环形队列实现，为了确保多消费者不同时取出同

一资源，需要使用一个互斥信号量 `mutex_buff` 来保护环形队列 `buffer`，使得变量 `buffer` 能够被多个消费者互斥地访问。

定义两个信号量分别为用于通知生产者生产资源的信号量和用于通知消费者消费资源的信号量。

```
int i = 0;
semaphore mutex_i;      // 互斥地访问变量 i
semaphore mutex_buff;   // 互斥信号量，用于互斥地访问缓冲池
semaphore sem_consumer; // 消费者信号量，大于 0 表示可以消费资源了
semaphore sem_producer; // 生产者信号量，大于 0 表示可以生产资源了
```

(3) 定义生产者进程

下面是生产者进程的实现代码，对于互斥信号量，通常在同一个进程中成对的出现，这里借助 C 语言的花括号和缩进更清晰地展现出互斥信号量之间的成对关系。首行为获取进程的 `pid` 用于进程的唯一标识符，方便控制台打印输出，然后便进入了进程循环。

首先需要等待缓冲区不满，使生产者能够顺利生产，当信号量 `sem_producer` 执行 `wait` 操作结束后，标识生产者可以生产，此时生产者需要互斥地访问缓冲池。获取到缓冲池的使用权后，还需要再互斥地访问变量 `i`，于是便进入了生产流程。生产结束后，需要释放互斥信号量 `mutex_i` 和 `mutex_buff`，否则将发生进程死锁，最后便可以通知消费者进程消费资源。

```
// 生产者
void producer() {
    int pid = task_get_current_id();
    while (1) {
        semaphore_wait(&sem_producer); // 等待缓冲区存在空闲空间，能够生产
        semaphore_wait(&mutex_buff);   // 取得缓冲池
        semaphore_wait(&mutex_i);      // 保护序号记录变量 i，互斥地生产
        buffer_enqueue(i);
        log_info("producer: %d    生产产品: %d", pid, i);
        i++;
        semaphore_signal(&mutex_i);    // 释放序号记录变量 i
        semaphore_signal(&mutex_buff);  // 释放缓冲池
        semaphore_signal(&sem_consumer); // 通知消费者可以消费了
    }
}
```

(4) 定义消费者进程

消费者进程和生产者进程的写法大致结构相似，首先需要等待缓冲区不空，存在可以消费的资源，然后取得缓冲池的访问权限，取出资源后释放缓冲区，通知生产者可以继续生产。

```
void consumer() {
    int pid = task_get_current_id();
    while (1) {
```

```
semaphore_wait(&sem_consumer); // 等待缓冲区存在空闲产品，能够消费
semaphore_wait(&mutex_buff); // 取得缓冲池
log_info("consumer: %d 消费产品: %d", pid, buffer_dequeue());
semaphore_signal(&mutex_buff); // 释放缓冲池
semaphore_signal(&sem_producer); // 通知生产者可以生产了
}
}
```

(5) 初始化信号量并创建进程

最后，需要初始化所有信号量的初值，互斥量又称 01 信号量，其初值为 1。生产者信号量指示了能够有多少空间用于供生产者生产，初始状态应当为缓冲区的最大长度。消费者信号量指示了能够消费多少资源，初始状态下应当是 0。生产者创建进程，使进程进入就绪态随时被操作系统调度到 CPU 上运行，开始生产和消费。如下代码创建了两个生产者和三个消费者。

```
void user_main() {
    semaphore_init(&mutex_buff, 1); // 互斥量
    semaphore_init(&mutex_i, 1); // 互斥量
    semaphore_init(&sem_producer, BUFF_SIZE); // 初始可以一次性生产满缓冲区
    semaphore_init(&sem_consumer, 0); // 初始不能消费
    for (int i=0;i<2;i++) task_create(producer, NULL); // 2 个生产者
    for (int i=0;i<3;i++) task_create(consumer, NULL); // 3 个消费者
}
```

(6) 运行测试

本用户态示例程序的运行结果如图 4.2 所示，首先定义了长度为 5 的缓冲区，接着创建三个任务进程，1 号任务为生产者，2 号任务为消费者，3 号任务用于实现生产产品 id 超过 100 后及时通知生产者消费者进程停止运行。起初，缓冲区为空，生产者开始生产产品，当生产至 4 号产品时，缓冲区满，生产者进程进入阻塞状态。随后进程调度器调度 CPU 至消费者进程时，消费者消费了 0 号产品，此时缓冲区不满，消费者通知生产者可以生产了，生产者进入就绪态，当消费者消费至 4 号产品后，队列中所有产品消费完毕，消费者进入阻塞状态。随后进程调度器调度 CPU 至生产者进程，生产者生产了产品 5，此时缓冲区队列不空，通知消费者进程进入就绪状态。如此往复循环，便可持续推进生产者，消费者进程之间协作完成生产消费任务。

```

[> producer_consumer
[INFO] user_producer_consumer.c:96: 基于信号量的生产者消费者演示启动
[INFO] user_producer_consumer.c:100: buffer size: 5
[DEBUG] task.c:94: task id 1 is created
[DEBUG] task.c:94: task id 2 is created
[DEBUG] task.c:94: task id 3 is created
/> [INFO] user_producer_consumer.c:61: producer: 1 生产产品: 0
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 1
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 2
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 3
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 4
[DEBUG] task.c:61: task id 1 is blocked
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 0
[DEBUG] task.c:68: task id 1 is ready
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 1
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 2
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 3
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 4
[DEBUG] task.c:61: task id 2 is blocked
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 5
[DEBUG] task.c:68: task id 2 is ready
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 6
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 7
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 8
[INFO] user_producer_consumer.c:61: producer: 1 生产产品: 9
[DEBUG] task.c:61: task id 1 is blocked
[INFO] user_producer_consumer.c:80: consumer: 2 消费产品: 5
[DEBUG] task.c:68: task id 1 is ready

```

图 4.2 运行结果

4.3 文件系统目录的递归遍历

文件系统目录树的递归遍历是一个经典的较为综合性的文件系统操作，本课题将模拟创建具有较为丰富的目录层次结构，并实现递归地遍历，格式化打印输出具有目录深度缩紧的字符串。

(1) 创建各级目录与文件

```

static void create_dir_tree() {
    FILE fp;
    f_mkdir("dir1"); // 创建目录 dir1
    f_mkdir("dir1/dir11"); // 在 dir1 中创建子目录 dir11
    // 在 dir1/dir11 中创建子文件 file111 和 file112
    f_open(&fp, "dir1/dir11/file111", FA_CREATE_ALWAYS | FA_WRITE);
    f_open(&fp, "dir1/dir11/file112", FA_CREATE_ALWAYS | FA_WRITE);
    f_mkdir("dir1/dir12"); // 后续代码同理
    f_open(&fp, "dir1/dir12/file121", FA_CREATE_ALWAYS | FA_WRITE);
    f_open(&fp, "dir1/dir12/file122", FA_CREATE_ALWAYS | FA_WRITE);
    f_mkdir("dir2");
    f_mkdir("dir2/dir21");
    f_open(&fp, "dir2/dir21/file211", FA_CREATE_ALWAYS | FA_WRITE);
    f_open(&fp, "dir2/dir21/file212", FA_CREATE_ALWAYS | FA_WRITE);
}

```

```

    f_mkdir("dir2/dir22");
    f_open(&fp, "dir2/dir22/file221", FA_CREATE_ALWAYS | FA_WRITE);
    f_open(&fp, "dir2/dir22/file222", FA_CREATE_ALWAYS | FA_WRITE);
}

```

(2) 递归遍历目录树

```

static void list_dir_tree(TCHAR *path, int depth) {
    DIR dir;  FILINFO fno;  TCHAR subpath[256];  FRESULT res;
    res = f_findfirst(&dir, &fno, path, "*"); // 首先列举指定路径下的所有文件
    while (res == FR_OK && fno.fname[0]) {
        int n = depth * 2; // 用于控制递归遍历的深度层级, 实现有层次地打印输出
        if (fno.fattrib == AM_DIR) {
            // 如果是目录则拼接路径后继续递归地遍历
            while (n--) printf(" ");
            printf("%s%s\\n", "", fno.fname);
            sprintf(subpath, "%s/%s", path, fno.fname);
            list_dir_tree(subpath, depth + 1);
        } else {
            // 如果是文件, 则直接输出
            while (n--) printf(" ");
            printf("%s\\n", fno.fname);
        }
        res = f_findnext(&dir, &fno);
    }
}

```

(3) 测试运行

```

const BYTE pdrv = 0;  FATFS fs;
disk_initialize(pdrv); // 磁盘初始化
make_partition(pdrv); // 磁盘分区
format_volume(); // 格式化磁盘
assert(f_mount(&fs, "0:", 0) == FR_OK); // 挂载磁盘
create_dir_tree(); // 创建目录树
list_dir_tree("", 0); // 递归地列举目录树
assert(f_mount(&fs, "", 0) == FR_OK); // 卸载磁盘

```

(4) 运行结果

运行结果输出如图 4.3 所示

```
[/>> ls
dir1
dir2
[/>> tree
dir1/
  dir11/
    file111
    file112
  dir12/
    file121
    file122
dir2/
  dir21/
    file211
    file212
  dir22/
    file221
    file222
[/>>
```

图 4.3 运行结果

4.4 Shell 命令操作接口的实现

本课题实现了一个简单的操作系统命令交互接口 Shell，仿照 Linux 操作系统的命令完成了总计 27 条命令，如图为所有 Shell 命令的注册入口函数，该函数实际上是由操作系统内核启动后创建的第一个子进程，后续启动的进程均为 shell 进程启动的子进程。本课题演示的 Shell 中完成了 echo 用于在终端打印输出指定字符串，clear 用于清屏；help 用于显示操作系统 Shell 中支持的所有命令；open, ls, cat, cd, touch, mkdir, tree, append, appendln, rm, rmdir 为文件管理相关命令；log, logset 为日志系统管理命令；exec, bat 为批处理运行相关命令；rand, demo, linkedlist, producer_consumer, filesystem, heap, calc, hanoi, maze 为在本操作系统之上编写的所有用户程序，其中 rand 用于打印一个随机数，calc 用于计算求解一个表达式字符串，hanoi 为一个汉诺塔演示程序，maze 为一个简单的走迷宫小游戏。


```
void user_shell() {
    _cmd_size = 0;
    running = true;
    register_command("echo", echo_cmd);
    register_command("help", help_cmd);
    register_command("ls", ls_cmd);
    register_command("open", open_cmd);
    register_command("cat", open_cmd);
    register_command("cd", cd_cmd);
    register_command("touch", touch_cmd);
    register_command("mkdir", mkdir_cmd);
    register_command("log", log_cmd);
    register_command("logset", logset_cmd);
    register_command("tree", tree_cmd);
    register_command("append", append_cmd);
    register_command("appendln", appendln_cmd);
    register_command("exec", exec_cmd);
    register_command("bat", bat_cmd);
    register_command("rm", rm_cmd);
    register_command("rmdir", rmdir_cmd);
    register_command("clear", clear_cmd);
    register_command("rand", rand_cmd);
    register_command("demo", user_demo);
    register_command("linkedlist", user_filesystem);
    register_command("producer_consumer", user_producer_consumer);
    register_command("filesystem", user_filesystem);
    register_command("heap", user_heap);
    register_command("calc", user_calc);
    register_command("hanoi", user_hanoi);
    register_command("maze", user_maze);

    user_init_script();
    run_shell();
}
```

图 4.4 Shell 命令操作接口

(1) heap 内存分配演示程序

如图 4.5 为一个内存分配演示程序，在堆内存空间中，目前总计分配了三段内存空间，段内存首地址分别为 0x8022da40, 0x8022fa40, 0x80232a40。本课题操作系统设计的内存分配器将内存区域以 4KB 大小为一块区域进行划分，这叫做分页内存管理，每页内存块使用 2bit 位做元数据信息的标注，如图 4.5 中显示了一些内存页面的元数据，左

侧的比特位标记了该内存区域是否已到达内存段的末尾，右侧的比特位标记了该内存区域是否已被占用。如图 4.5 中分别分配了三段内存区域，分别占用了 0, 1 号内存页，2, 3, 4 号内存页，5 号内存页，其内存页元数据信息在图中上方所示。

```

01 11 01 01 11 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
page 0 - 1 used, size: 2, addr: 0x8022da40
page 2 - 4 used, size: 3, addr: 0x8022fa40
page 5 - 5 used, size: 1, addr: 0x80232a40
选择待释放的内存地址：
0. page 0 - 1 used, size: 2 pages, addr: 0x8022da40
1. page 2 - 4 used, size: 3 pages, addr: 0x8022fa40
2. page 5 - 5 used, size: 1 pages, addr: 0x80232a40
3. 退出
Use 'w', 's' to move choice
Use 'q' to exit
Use 'Enter' to run choice

```

图 4.5 内存分配器演示程序

(2) 表达式求解器

如图 4.6 为表达式求解器，它将常用的中缀表达式利用栈先求解转化为后缀表达式，再使用栈完成后缀表达式的求解，从而实现对一个中缀表达式的计算求解，这是数据结构中经典的一个案例，在本操作系统中，主要涉及了字符串处理与栈空间分配的相关内容。

```

[/> calc -v 7+9-(8*6+8)
逆波兰表达式转换结果：7 9 + 8 6 * 8 +-
计算结果：-40
[/> 

```

图 4.6 表达式求解器

(3) 汉诺塔小程序

如图 4.7 为 3 层汉诺塔的求解程序运行结果，是算法设计中递归程序设计的一个常见的经典教学案例，它的代码如下：

```

void hanoi(int n, char a, char b, char c) {
    if (n == 1) {
        printf("%d: %c -> %c\n", hanoi_i++, a, c);
    } else {
        hanoi(n - 1, a, c, b);
        printf("%d: %c -> %c\n", hanoi_i++, a, c);
    }
}

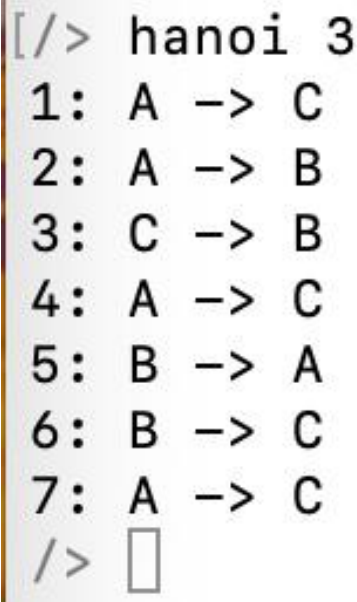
```



```

    hanoi(n - 1, b, a, c);
}
}

```



```

[/> hanoi 3
1: A -> C
2: A -> B
3: C -> B
4: A -> C
5: B -> A
6: B -> C
7: A -> C
/> █

```

图 4.7 汉诺塔小程序

任何递归算法最终均能够转化为栈进行实现，事实上 C 语言的运行环境必备的一个运行条件就是栈空间，一次函数调用本质上就是在栈上 push 一个栈帧，函数结束之后将 pop 出一个栈帧。本课题操作系统在程序引导程序中，为内核的 C 代码运行提供了 80KB 的字节空间作为内核栈空间，在代码 start.S 中定义。对于用户态应用程序，用户程序作为一个进程运行在操作系统中，它所占用的栈空间需要由进程管理模块创建进程时为其提供。

若调用函数 `hanoi(3, 'A', 'B', 'C')` 则相应等价的汇编代码如下：

```

li      a3, 67 # 寄存器 a3 加载立即数 67，即 C 的 ascii 码
li      a2, 66 # 寄存器 a2 加载立即数 66，即 B 的 ascii 码
li      a1, 65 # 寄存器 a1 加载立即数 65，即 A 的 ascii 码
li      a0, 3  # 寄存器 a0 加载立即数 3
call    hanoi # 调用符号 hanoi

```

实际上 `call` 指令是汇编伪指令，最终将编译为直接对于机器语言的汇编跳转指令，它的作用就是置 PC 寄存器的值为目标函数地址，使程序执行流发生改变。

(4) 走迷宫小游戏

如图 4.8 为走迷宫小游戏，是算法设计中深度优先搜索的一个经典且有趣的教学案例，该程序具备一定的可玩性，迷宫生成具备一定的随机性，由于本操作系统并未实现随机数发生器的驱动程序，故它使用了基于线性同余算法来进行伪随机数的生成。线性同余算法递推公式如下，其中 A, B, M 均为常数：

$$x_{n+1} = (A \times x_n + B)(mod M)$$

其中 x_0 为用户初始化的随机数种子，通常选取当前时间戳作为随机数种子。然后每一次调用 rand 函数将不断地得到序列 x_1, x_2, x_3, \dots 作为随机数序列。

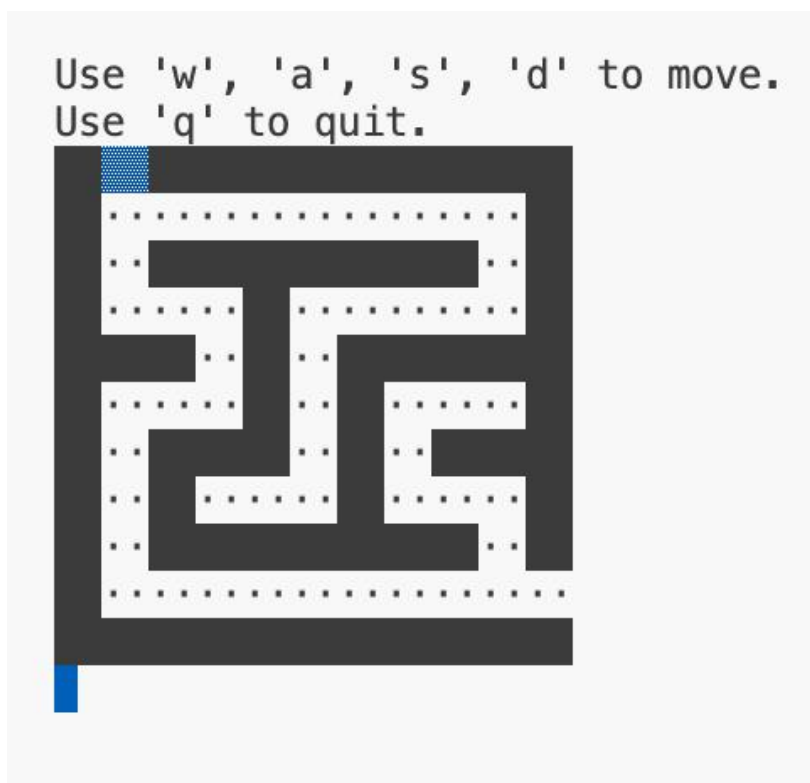


图 4.8 走迷宫小游戏

5 总结

(1) 总结

本课题研究旨在实现了一个小型的、可用于教学使用的嵌入式操作系统。通过采用软件工程的思想,本课题进行了较完善的需求开发,逐步对各个用例场景进行用例建模,最终得到了完整细致的整体模块图,为后续具体的实现提供了清晰的路线。

本课题使用了开源的 GNU 的工具链来构建,其中包括 GNU GCC 作为 C 语言编译器,GNU Binutils 工具集作为汇编器和二进制文件操作工具,Make 工具作为项目构建工具,编写了大量的 Makefile 代码来组织项目结构,完成项目构建工作,以及使用 Git 作为项目开发的版本控制工具;本课题完成后的操作系统项目最终运行在 QEMU 模拟硬件平台。

在实现部分,本课题阐述了一个 C 语言和汇编语言的混合项目到最终被 QEMU 所加载的可执行文件完整的编译流程。本课题使用汇编语言编写引导程序,为 C 语言的执行提供了必要的运行环境,在 C 语言中,编写了串口输出的驱动程序,最终成功在 QEMU 模拟的 RISC-V 架构机器下输出了 Hello, World 并重定向至宿主机的终端,提供了方便的日志输出,为后续的程序开发和调试工作带来了一定的便捷性。本课题还完成了一个简页的页级内存分配器,实现了 malloc 和 free 的内存分配相关 API 接口。此外,本课题完成了进程管理,实现了 PCB 结构体,实现了进程栈的初始化,进程上下文的初始化,进程的创建与销毁,进程的状态转换,进程的 CPU 让权。实现中断管理,并基于定时器中断完成了一个简页的时间片轮转的抢占式进程调度器。本课题还完成了进程同步,基于 CAS 机制和硬件提供的 CAS 原子指令实现了自旋锁,基于自旋锁实现了信号量的机制,可进行更为复杂的并发程序的编写。基于 FatFs 项目,以一块内存区域模拟实际的物理磁盘,完成了 FAT32 文件系统的移植工作。本课题基于 FatFs,完整地展示了从一块裸磁盘开始进行分区、FAT32 文件系统的格式化、创建文件、写入文件、关闭文件、获取文件属性、打开并读取文件、创建目录、目录遍历等文件系统相关的操作。

在经典案例的复现部分,基于本课题所完成的操作系统,实现了多个经典的数据结构和操作系统案例。例如,利用内存分配器复现了数据结构课程上的链表数据结构,利用进程管理与进程同步机制复现了操作系统课程上经典的生产者—消费者问题,利用了 FatFs 项目实现了文件系统目录树的递归遍历。通过对这些经典案例的复现,进一步证明了本课题操作系统的具备一定的可用性与可教学性。

本课题所完成的嵌入式操作系统代码相当精简易读,并不具有商业价值,仅供教学目的所使用。然而,如果需要将本课题的操作系统改造成非嵌入式操作系统,就需要考虑更多的安全机制和性能问题。例如需要借助 MMU 硬件完成虚拟地址映射机制,实现进程的内存隔离。此外,还需要划分出清晰的用户态和内核态的概念并通过系统调用在用户态完成系统提供的服务。实际上,RISC-V 架构的 CPU 级别分为三个等级模式,分

别为 Machine、Supervisor、User 模式。用户态应用程序应当运行 User 模式，操作系统应当运行在 Machine 或 Supervisor 模式，用户态应用程序运行在更高的特权级，通过借助硬件提供的 ecall 指令来实现 CPU 迁移到更低的特权级进入内核态，内核处理完毕系统调用之后，借助 eret 指令实现 CPU 迁移到更高的特权级返回用户态。

目前本课题的操作系统目前并未实现严格的系统调用机制，系统内核代码与用户代码均在 Machine 模式下运行，严格意义上讲用户代码调用系统内核服务应该通过系统调用完成的，但是本课题对其进行了大大简化，直接以一个普通的函数调用进行实现。这种简化在嵌入式场景的实时操作系统中实际上是一种很常见的操作，因为它能保证较低的性能开销和一定的实时性。在本课题中，这种方式也能够确保操作系统代码的简洁性和易读性，无需学习者过度关注 CPU 的特权级，特权指令，特权级切换等繁琐的机制，有利于减少学习者的心智负担，更加容易理解阅读代码实现。

总而言之，本课题旨在设计实现一套非常适合用于教学的一款嵌入式操作系统，其设计之初便放弃了一定的使用价值，牺牲了一定的安全性、稳定性、高效性，但是保障了架构简洁、代码易读，相当适合操作系统的初学者探究操作系统内部的具体实现原理，而不再是面向一个黑箱去学习研究。

致谢

在我完成本论文的过程中，得到了许多人的帮助和支持，我在此向他们表示衷心的感谢。

首先，我要感谢我的指导教师张成姝老师，她在整个研究过程中给予了我无私的指导和帮助。她的专业知识和经验对我完成论文起到了至关重要的作用。在整个研究过程中给予了我无私的指导和支持，在研究思路、程序设计、论文格式等方面给予了宝贵的指导和建议。她耐心严谨的治学态度和对科研的热情感染了我，让我受益匪浅。

其次，我要感谢我的同学和朋友们，他们在我研究的过程中提供了宝贵的建议和支持。我还要感谢我的家人，他们一直以来给予了我无条件的关爱和支持是我不断前进的动力源泉。。

在此，我要感谢所有为本课题提供了建议和支持的人。

参考文献

- [1] O. M. Ritchie and K. Thompson, "The UNIX time-sharing system," in The Bell System Technical Journal, vol. 57, no. 6, pp. 1905-1929, July-Aug. 1978, doi: 10.1002/j.1538-7305.1978.tb02136.x.
- [2] Bhat W A, Quadri S M K. Review of FAT Data Structure of FAT32 file system[J]. Oriental Journal of Computer Science & Technology, 2010, 3(1): 161-164.
- [3] Leijen D, Zorn B, de Moura L. Mimalloc: Free list sharding in action[C]//Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1 – 4, 2019, Proceedings 17. Springer International Publishing, 2019: 244-265.
- [4] Cox R, Kaashoek M F, Morris R. Xv6, a simple Unix-like teaching operating system[J]. 2022-11-05]. <https://pdos.csail.mit.edu/6.1810/2022/xv6/book-riscv-rev3.pdf>, 2022.
- [5] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1", Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 31, 2016.
- [6] 陈海波. 现代操作系统: 原理与实现[M]. 机械工业出版社, 2020.
- [7] 张成姝. 操作系统教程 (第二版) [M]. 清华大学出版社, 2019.
- [8] 曹成. 嵌入式实时操作系统 RT-Thread 原理分析与应用[D]. 济南: 山东科技大学, 2011.
- [9] 邱伟. 嵌入式实时操作系统 RT-Thread 的设计与实现[D]. 电子科技大学, 2007.
- [10] 胡振波. RISC-V 架构与嵌入式开发快速入门[M]. 人民邮电出版社, 2019.
- [11] 林金龙. 深入理解 RISC-V 程序开发[M]. 北京航空航天大学出版社, 2021.
- [12] 于渊. Orange'S: 一个操作系统的实现[M]. 电子工业出版社, 2009.
- [13] 张依依. RISC-V 2023: 难点也是突破点[N]. 中国电子报, 2023-01-13(008). DOI:10.28065/n.cnki.ncdzb.2023.000056.

附录

附录见所附软盘或光盘。