

# 目 录

1 选题背景.....	3
2 研究目的与意义.....	4
3 设计原则和总体规划.....	5
3.1 设计原则.....	5
3.2 总体规划.....	6
4 系统设计.....	8
4.1 需求分析.....	8
4.2 总体设计结构.....	10
4.3 功能模块详细设计.....	11
4.3.1 引导和系统加入.....	11
4.3.2 UART 串口设备 .....	12
4.3.3 内存管理.....	16
4.3.4 Trap 和 Exception（异常） .....	17
4.3.5 硬件定时器和软件定时器.....	18
4.3.6 抢占式多任务.....	20
4.3.7 锁.....	22
4.3.8 内核态切换.....	22
5. 编码实现.....	25
5.1 引导和系统加入.....	25
5.2 UART 串口设备 .....	25
5.3 内存管理.....	28
5.4 Trap 和 Exception（异常） .....	33
5.5 硬件定时器和软件定时器.....	35
5.6 抢占式多任务.....	36
5.7 锁.....	38
5.8 内核态切换.....	39
6.测试.....	41

6.1 系统 IO 测试 .....	41
6.2 内存管理测试.....	41
6.3Trap 和异常测试 .....	41
6.4 定时器测试.....	43
6.5 多任务测试.....	43
6.6 锁测试.....	45
6.7 内核态的切换和系统调用.....	46
7. 项目总结.....	47

# 1 选题背景

随着计算机技术的不断发展，操作系统在计算机系统中扮演着重要的角色。为了更好地理解和掌握操作系统的原理和设计，本项目旨在使用 C 语言和汇编语言编写一个支持 RISC-V 指令集架构的 32 位单核操作系统。该操作系统旨在实现以下关键功能：

**引导和系统初始化：**设计一个引导程序，将操作系统加载到内存中，并正确初始化系统资源。

**任务调度和多任务支持：**实现上下文切换机制和协作式多任务调度，使多个任务能够共享处理器时间。

**中断和异常处理：**处理系统中的中断和异常事件，确保系统能够正确响应和处理各种异常情况。

**内存管理和资源分配：**设计内存管理机制，包括内存分配和释放，以有效管理系统内存资源。

**外部设备支持：**处理外部设备中断，使系统能够及时响应外部设备事件，并进行相应的处理。

**定时器和时间管理：**利用硬件定时器实现时间片轮转调度算法，确保任务按照预定的时间间隔执行。

通过该项目的实施，我们将深入了解操作系统的内部工作原理，提高对计算机系统的理解，并具备自主设计和实现基本操作系统的能力。同时，该操作系统也为后续的系统扩展和功能开发提供了基础。

## 2 研究目的与意义

本项目的研究目的是深入了解计算机系统，并通过开发一个支持 RISC-V 指令集架构的 32 位单核操作系统来实现这一目标。具体的研究目的和意义如下：

**理解操作系统原理：**通过开发操作系统，我们将深入研究操作系统的各个组成部分，包括任务调度、内存管理、中断处理等，从而全面理解操作系统的工作原理和设计思想。

**掌握计算机系统开发技术：**通过实践开发操作系统，我们将掌握操作系统开发所需的技术和工具，包括编程语言（C 语言和汇编语言）、调试工具、代码组织和模块设计等，提高我们在计算机系统领域的实际能力。

**培养系统级思维和解决问题的能力：**操作系统是计算机系统的核心组成部分，开发操作系统需要系统级思维和解决问题的能力。通过这个项目，我们将培养这些能力，能够从整体角度考虑问题，并设计出高效、可靠的解决方案。

**加强对计算机体系结构的理解：**通过开发基于 RISC-V 指令集架构的操作系统，我们将更深入地了解该体系结构的特点、指令集和编程模型，提高对计算机体系结构的理解和应用能力。

**为后续研究和项目奠定基础：**操作系统是计算机科学领域的重要研究方向，通过这个项目的实施，我们将为进一步研究和开发更复杂、功能更强大的操作系统奠定基础，为未来的研究和项目提供有力支持。

通过深入研究计算机系统并开发操作系统，我们能够全面理解和掌握计算机系统的工作原理，提高实际能力，并为未来的研究和项目打下坚实基础。

## 3 设计原则和总体规划

### 3.1 设计原则

设计操作系统时，可以遵循以下设计原则：

**模块化和分层设计：**将操作系统划分为多个模块和层次，每个模块负责特定的功能。模块化设计可以降低系统复杂性，提高可维护性和可扩展性。通过定义清晰的接口和协议，模块之间可以进行交互和通信。

**最小特权原则：**每个模块和组件应该具有最小的权限和访问权限，只能访问其所需的资源 and 功能。这样可以减少潜在的安全漏洞和错误，并提高系统的安全性和稳定性。

**开放性和可扩展性：**设计操作系统时要考虑到未来的扩展和功能增加。提供开放的接口和机制，使其他开发者能够方便地添加新的功能、设备支持或定制化扩展。

**简单性和可理解性：**操作系统的设计应该尽可能简单和易于理解。避免过度复杂的设计和实现，以降低错误和维护成本，并方便其他开发者理解和修改系统。

**性能优化：**在设计和实现过程中，要考虑系统的性能优化。合理利用硬件资源、优化算法和数据结构、减少上下文切换等，以提高系统的响应速度和效率。

**可移植性：**考虑到不同硬件平台和处理器架构的差异，设计操作系统时应考虑可移植性。

**容错性和可靠性：**操作系统应具备容错性，能够正确处理和恢复各种异常情况，如硬件故障、内存溢出等。设计合适的异常处理机制和容错策略，确保系统的稳定性和可靠性。

**安全性和权限管理：**考虑到操作系统涉及到用户数据和敏感信息的处理，安全性是一个重要的设计原则。设计合适的安全性措施和权限管理机制，确保系统的安全性和数据保护。

**调试和测试支持：**为了方便调试和测试操作系统，设计时应考虑提供相应的调试和测试支持，如日志记录、调试接口、模拟器等。这样可以帮助开发者验证系统的正确性和性能，并进行故障排查。

## 3.2 总体规划

**调试和测试支持：**为了方便调试和测试操作系统，设计时应考虑提供相应的调试和测试支持，如日志记录、调试接口、模拟器等。这样可以帮助开发者验证系统的正确性和性能，并进行故障排查。

总体规划是指在设计 and 开发操作系统时的整体规划和策略。以下是一些可能的总体规划考虑：

**功能规划：**确定操作系统需要支持的功能和特性。这可能包括任务调度、内存管理、文件系统、设备驱动程序、网络支持等。根据需求和目标，定义功能模块的划分和接口。

**架构规划：**确定操作系统的整体架构。这可能涉及到选择适当的系统架构，如微内核、宏内核、外核等。确定各个模块的关系和交互方式，包括模块间的通信机制和数据传递方式。

**硬件支持规划：**考虑操作系统所运行的目标硬件平台和处理器架构。确定操作系统对硬件的要求和支持范围，以确保操作系统能够在目标硬件上正确运行并充分利用硬件资源。

**接口规划：**定义操作系统与外部世界的接口和交互方式。这可能包括系统调用接口、设备驱动程序接口、用户界面等。确保接口的一致性和兼容性，以便应用程序和外部设备能够与操作系统进行有效的通信。

**开发流程规划：**制定操作系统的开发流程和项目计划。确定开发阶段、时间表和里程碑，分配任务和资源。制定适当的开发和测试方法，确保高质量的操作系统交付。

**调试和测试规划：**规划操作系统的调试和测试策略。确定合适的调试工具和技术，确保开发者能够有效地调试和排除错误。设计全面的测试计划，包括单元测试、集成测试和系统测试，以验证操作系统的正确性和性能。

**文档和支持规划：**规划操作系统的文档编写和用户支持策略。编写清晰、详细的文档，包括用户手册、开发者文档和 API 文档。提供相应的支持渠道，如论坛、邮件列表等，以使用户和开发者能够得到及时的支持和反馈。

**部署和维护规划：**考虑操作系统的部署和维护策略。确定操作系统的安装和

配置过程，确保用户能够轻松地部署和启动操作系统。制定维护计划，包括更新和修复操作系统的流程和策略。

这是一些可能的总体规划考虑，实际的总体规划应根据具体项目需求和目标进行进一步的定制和调整。总体规划的目的是确保操作系统的设计和开发能够按照有序和合理的方式进行，以达到预期的目标和交付高质量的操作系统。

## 4 系统设计

### 4.1 需求分析

本项目旨在使用 C 和汇编语言开发一个基于 RISC-V 指令集架构的 32 位支持单核的操作系统。该操作系统将实现以下功能：

1. 引导和系统入口：

操作系统将具有引导功能，能够正确加载并启动操作系统。系统入口提供操作系统的初始化和配置。

2. UART 串口设备：

操作系统将支持 UART 串口设备，用于与外部设备进行通信。通过串口，用户可以进行输入输出交互，例如命令行界面。

3. 内存管理：

操作系统将实现内存管理机制，包括虚拟内存管理和物理内存管理。通过内存管理，实现内存的分配、释放和保护，确保不同任务之间的内存隔离和安全性。

4. 上下文切换和协作式多任务：

操作系统将支持多任务机制，实现任务的切换和调度。通过上下文切换，实现任务间的切换和资源共享，以实现协作式多任务。

5. Trap 和 Exception：

操作系统将处理和管理 RISC-V 的陷阱和异常。它能够捕获和处理不同类型的陷阱和异常，以实现错误处理和系统保护。

6. 外部设备中断：

操作系统将支持外部设备中断的处理。当外部设备发生中断时，操作系统能够及时响应中断请求，并正确处理中断事件。

7. 硬件定时器：

操作系统将利用硬件定时器实现时间管理和延时功能。通过硬件定时器，操作系统可以进行时间片轮转调度、任务超时等操作。



8. 抢占式多任务：

操作系统将实现抢占式多任务机制，以提高系统的响应能力和效率。当有高优先级任务出现时，操作系统能够抢占当前任务并进行任务切换。

9. 任务同步和锁：

操作系统将提供任务同步和互斥锁机制，以实现任务间的协调和资源访问的互斥。通过任务同步和锁，确保任务之间的正确执行和数据的一致性。

10. 软件定时器：

操作系统将支持软件定时器，用于实现定时任务和周期性任务。通过软件定时器，操作系统可以定期执行特定的任务或操作。

11. 内核态切换：

操作系统将支持内核态和用户态之间的切换，以实现系统资源的保护和权限管理。

此外，操作系统将提供一个简单的命令行界面，使用户能够通过命令行输入和执行简单的操作。通过命令行界面，用户可以与操作系统进行交互和管理。

## 4.2 总体设计结构

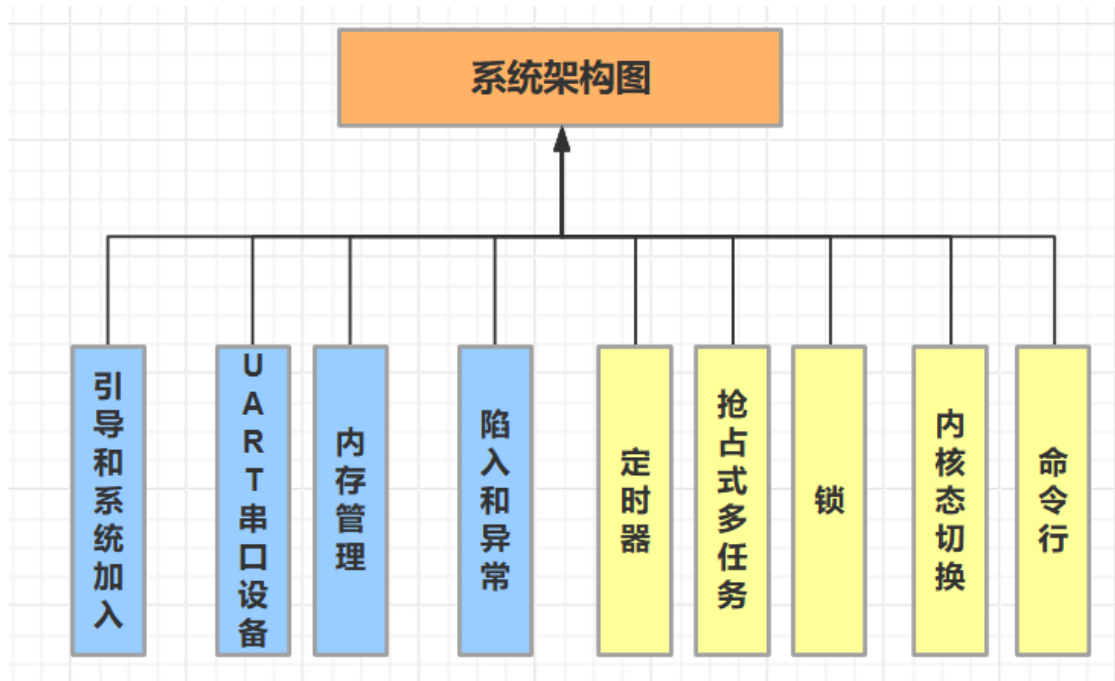


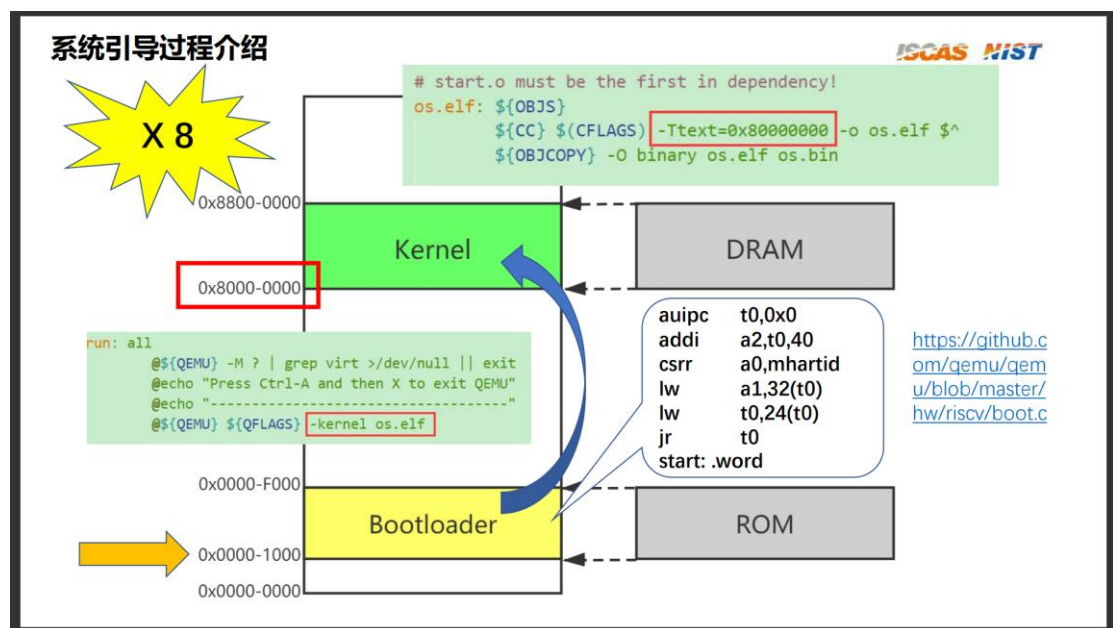
图 4-1 系统架构图

## 4.3 功能模块详细设计

### 4.3.1 引导和系统加入

设计一个引导程序，能够在计算机启动时加载操作系统到内存中，并正确初始化系统资源。

QEMU-virt 地址映射



4-2 系统引导过程

1. 判断当前 hart 是不是第一个 hart
2. 初始化栈
3. 跳转到 C 语言的执行环境

## Control and Status Registers (CSRs)

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

【参考 2】Table 1.1: RISC-V privilege levels.

- 除了所有 Level 下都可以访问的通用寄存器组之外，每个 Level 都有自己对应的一组寄存器。
- 高 Level 可以访问低 Level 的 CSR，反之不可以。
- ISA Specification（“Zicsr”扩展）定义了特殊的 CSR 指令来访问这些 CSR。

图 4-3 进入系统相关的寄存器

### 4.3.2 UART 串口设备

#### • UART 的特点

串行：相对于并行，串行是按位来进行传递，即一位一位的发送和接收。波特率(baud rate)，每秒传输的二进制位数，单位为 bps(bit per second)。

异步：相对于同步，异步数据传输的过程中，不需要时钟线，直接发送数据，但需要约定通讯协议格式。

全双工：相对于单工和半双工，全双工指可以同时进行收发两方向的数据传递

#### • UART 的通讯协议

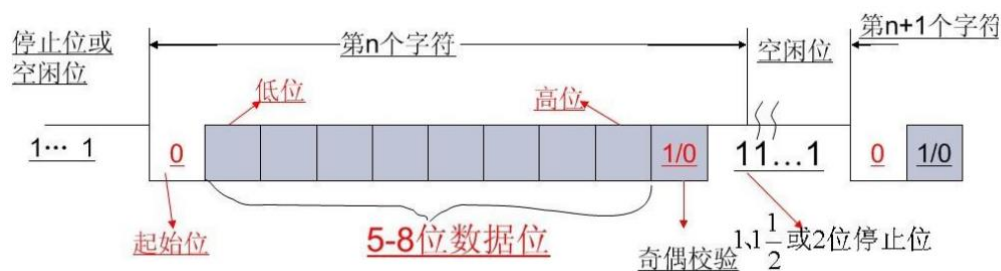


图 4-4 UART 的通讯协议图

空闲位：总线处于空闲状态时信号线的状态为 '1' 即高电平。

- 起始位：发送方要先发出一个低电平 '0' 来表示传输字符的开始。
- 数据位：起始位之后就是要传输的数据，数据长度（word length）可以是 5/6/7/8/9 位，构成一个字符，一般都是 8 位。先发送最低位最后发送最高位。
- 奇偶校验位（parity）：串口校验分几种方式：
  - ü 无校验（no parity）
  - ü 奇校验（odd parity）：确保传输数据（包含校验位）中 1 的个数为奇数。即：如果传输数据位中 1 的个数是偶数，则校验位取值为“1”（补足使得总的 1 的个数为奇数），否则如果传输数据位 1 的个数是奇数，则校验位取值为“0”。
  - ü 偶校验（even parity）：确保传输数据（包含校验位）中 1 的个数为偶数。即：如果传输数据位中 1 的个数是奇数，则校验位取值为“1”（补足使得总的 1 的个数为偶数），否则如果传输数据位 1 的个数是偶数，则校验位取值为“0”。
  - ü mark parity：校验位始终为 1
  - ü space parity：校验位始终为 0
- 停止（stop）位：数据结束标志，可以是 1 位，1.5 位，2 位 的高电平

• NS16550a 编程接口介绍

UART 设备映射的地址是 0x100000000

```
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 4-5UART 设备映射的地址图

A2	A1	A0	READ MODE	WRITE MODE
0	0	0	• Receive Holding Register	• Transmit Holding Register
0	0	0	N/A	• LSB of Divisor Latch when Enabled
0	0	1	N/A	• Interrupt Enable Register
0	0	1	N/A	• MSB of Divisor Latch when Enabled
0	1	0	• Interrupt Status Register	• FIFO control Register
0	1	1	N/A	• Line Control Register
1	0	0	N/A	• Modem Control Register
1	0	1	• Line Status Register	N/A
1	1	0	• Modem Status Register	N/A
1	1	1	• Scratchpad Register Read	• Scratchpad Register Write

图 4-6NS16550a 编程接口表

- NS16550a 的初始化

初始化设备需要设置波特率

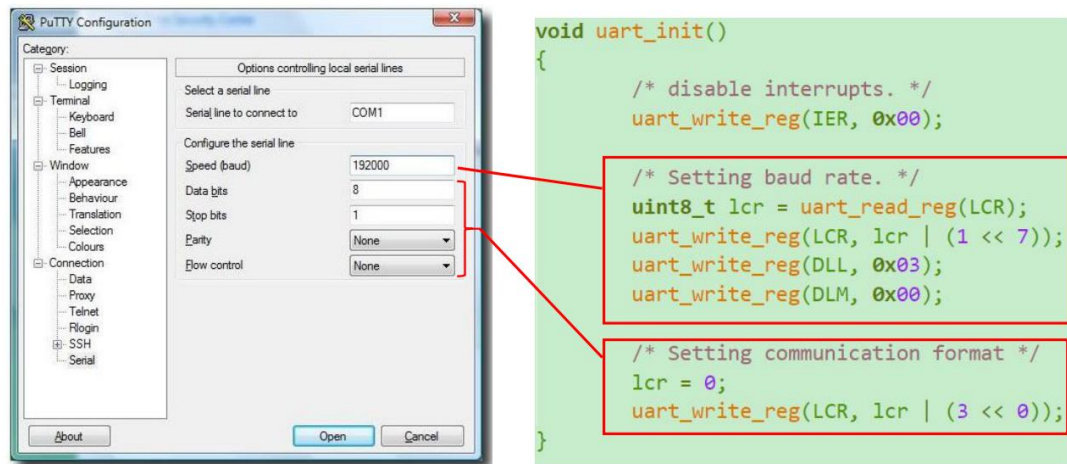


图 4-7 设置波特率

### 4.3.3 内存管理

对内存进一步的管理，实现动态的分配和释放。

实现 Page 级别的内存分配和释

- 1 自动管理内存 - 栈 (stack)
- 2 静态内存 - 全局变量/静态变量
- 3 动态管理内存 - 堆 (heap)

内存映射

内存映射表 (Memory Map)

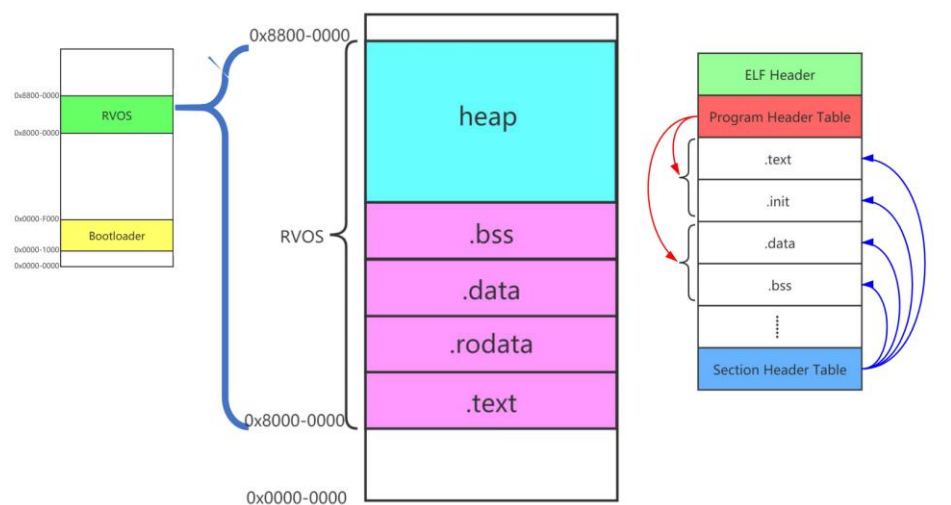


图 4-8 内存映射表图

基于 Page 实现动态内存

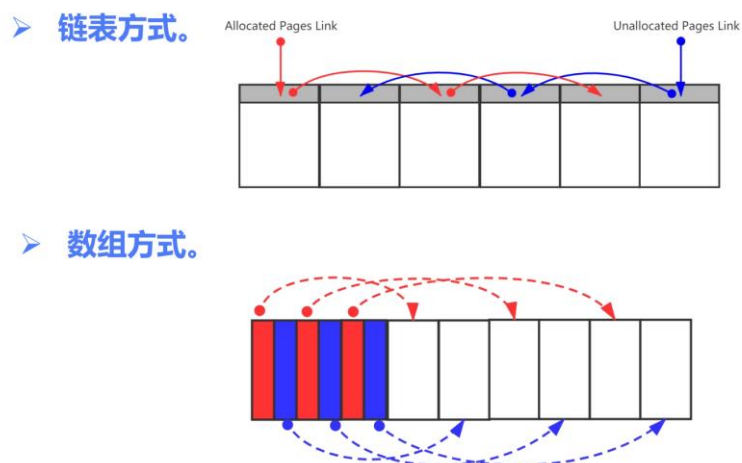


图 4-9 内存管理数据结构的两种方式

本项目使用的是数组方式



### 4.3.4 Trap 和 Exception（异常）

#### 1. 涉及的寄存器

寄存器	用途说明
mtvec (Machine Trap-Vector Base-Address)	它保存发生异常时处理器需要跳转到的地址。
mepc (Machine Exception Program Counter)	当 trap 发生时, hart 会将发生 trap 所对应的指令的地址值 (pc) 保存在 mepc 中。
mcause (Machine Cause)	当 trap 发生时, hart 会设置该寄存器通知我们 trap 发生的原因。
mtval (Machine Trap Value)	它保存了 exception 发生时的附加信息: 譬如访问地址出错时的地址信息、或者执行非法指令时的指令本身, 对于其他异常, 它的值为 0。
mstatus (Machine Status)	用于跟踪和控制 hart 的当前操作状态 (特别地, 包括关闭和打开全局中断)。
mscratch (Machine Scratch)	Machine 模式下专用寄存器, 我们可以自己定义其用法, 譬如用该寄存器保存当前在 hart 上运行的 task 的上下文 (context) 的地址。
mie (Machine Interrupt Enable)	用于进一步控制 (打开和关闭) software interrupt/timer interrupt/external interrupt
mip (Machine Interrupt Pending)	它列出目前已发生等待处理的中断。

图 4-10 Tarp 相关的寄存器

#### 2. Trap 处理流程

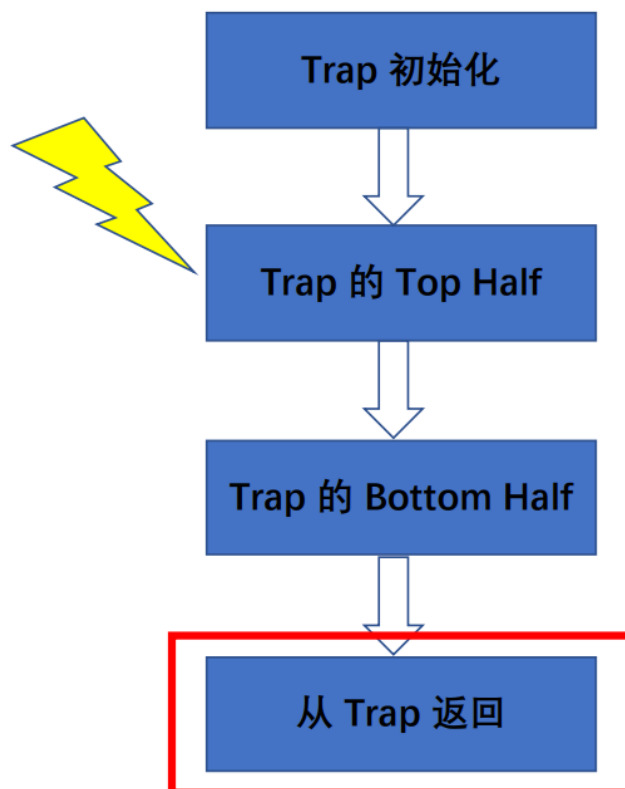


图 4-11 Trap 处理流程图

Trap 初始化阶段，先设置在 Machine 模式下 trap-vector 的基地址，

Trap 的 topHalf 阶段，mstatus 寄存器记录着系统当前模式和上一个模式的模式状态和中断情况。当 Trap 发生时，Hart 自动执行如下状态转换，将当前的模式的状态和中断保存起来，切换到 Machine 模式，关闭中断。并把中断的信息写入修改的寄存器

Trap 的 BottomHalf 阶段，系统会保存（save）当前控制流的

上下文信息（利用 mscratch），再调用 C 语言的 trap handler，根据中断表进行中断的处理，处理完后恢复中断前（restore）上下文的信息，执行 MRET 指令返回到 trap 之前的状态。

## 4.3.5 硬件定时器和软件定时器

### 1. CLINT 编程接口

RISC-V 规范规定，CLINT 的寄存器编址采用内存映射方式。

具体寄存器编址采用  $\text{base} + \text{offset}$  的格式，且 base 由各个特定 platform 自己定义。针对 QEMU-virt，其 CLINT 的设计参考了 SFIVE，base 为 0x2000000

### 2. 总体流程

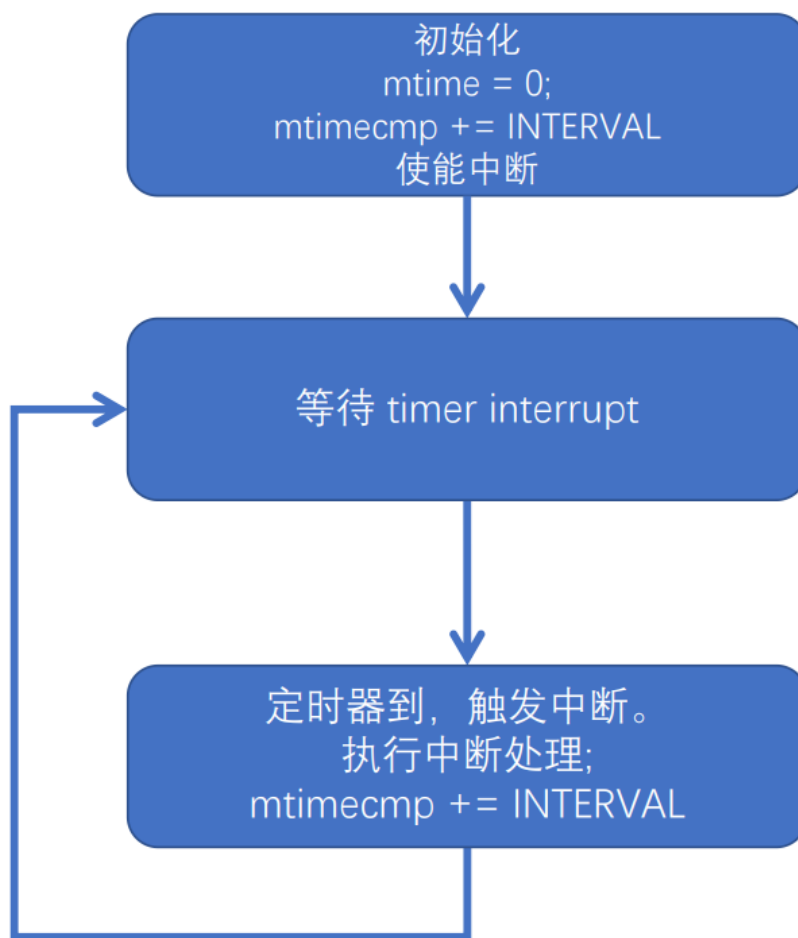


图 4-12 硬件定时器处理流程图

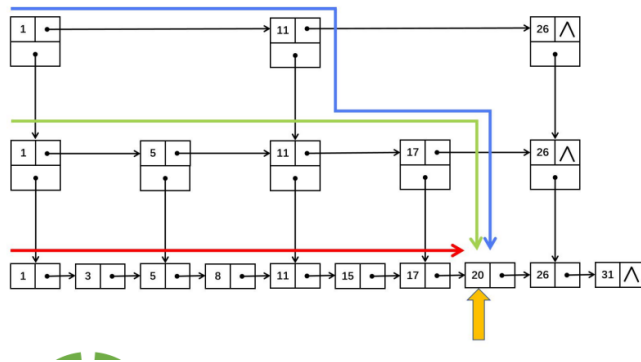
初始化阶段，`mtime` 寄存器通电自动为 0，开始递增，通过不断的设置 `mtimecmp` 寄存器的值，每次 `mtime` 的值超过 `mtimecmp`，会向系统发出一个定时器中断，中断以后，我们再重新设置 `mtimecmp` 的值，不断往复。

### 3. 自定义软件定时器

硬件定时器的数量受限于硬件。很难增加其数量，而通过软件模拟可以轻易实现任意数量的定时器。

软件模拟硬件，首先我们基于一个硬件定时器的值设置为系统的基准时钟，设置多个时间节点作为“闹钟”，每次系统的基准时钟改变，都遍历所有的“闹钟”看看是否有定时器时间到了，如果有，则执行该定时器的触发函数。

## ➤ 跳表 (Skip List) 算法



$O(\log(n))$

图 4-13 软件定时器的优化方向。

## 4.3.6 抢占式多任务

### 1. 抢占式多任务切换的触发

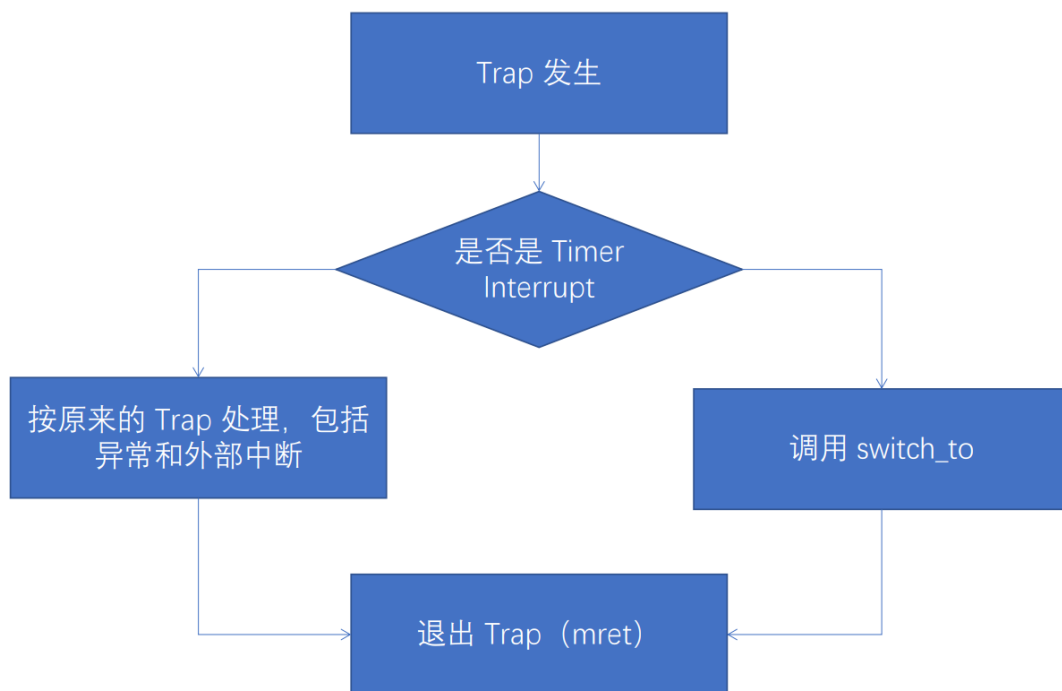


图 4-14 抢占式多任务切换的触发流程图

当 Trap 发生时，查看是否是定时器中断，如果是则调用任务切换的系统调用，如果不是，则进行其中断的相关处理。

## 抢占式多任务的设计

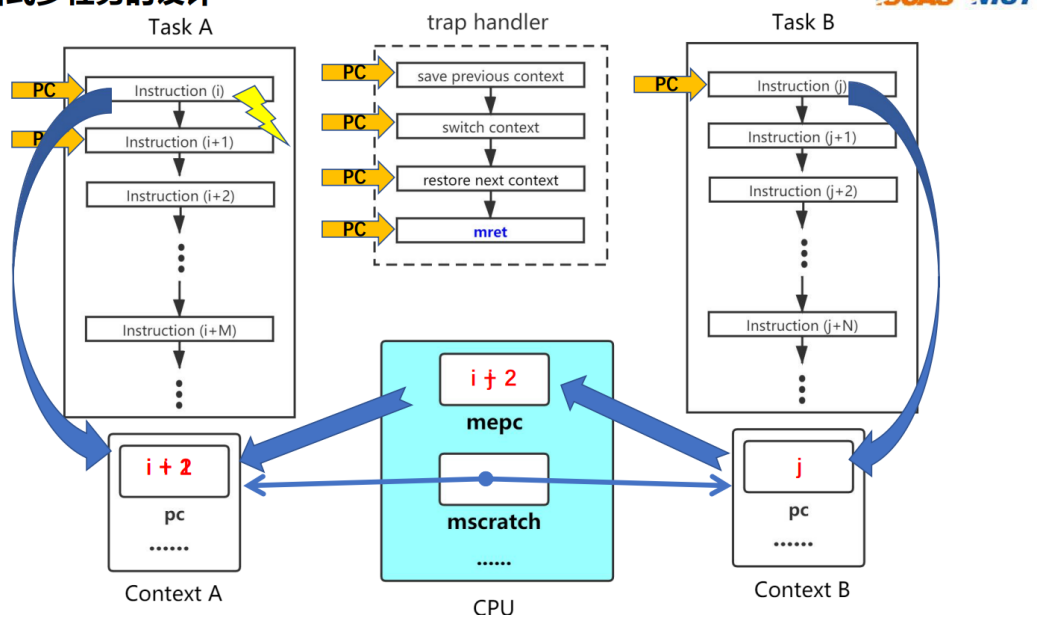


图 4-15 抢占式的具体细节

有 2 个特殊的寄存器：

**mepc**：记录着当前控制流的当前指令的地址

**mscratch**：是一个自定义寄存器，这里让他记录当前控制流的上下文基地址。

**pc**：记录本上下文的执行到的指令的地址。

任务的切换正是通过改变这 2 个寄存器和任务上下文切换函数实现的。

Trap 发生时，无论指令执行到哪，都会保存当前上下文到内存，进行 trap 相应的处理函数，处理完后恢复原先的上下文。

任务切换的原理：

在 Trap 处理函数中改变 pc 的值，使得 tarp 结束后恢复的是另一个任务，从而达到任务切换。

## 4.3.7 锁

### 1. 中断锁

本系统由于是单核系统，而锁的意义是为在并发的程序中保护共享资源被合理使用。在单核情况下，如果关掉中断，则可以保证共享资源同一时间只能被一个程序使用。

```
9-lock > lock.c > ...
#include "os.h"

int spin_lock()
{
    w_mstatus(x: r_mstatus() & ~MSTATUS_MIE); // 11111111011, 写0, 关闭中断
    return 0;
}

int spin_unlock()
{
    w_mstatus(x: r_mstatus() | MSTATUS_MIE); // 0000000100 写1, 开启中断
    return 0;
}
```

图 4-16 中断锁

### 2. 自旋锁

显然，通过关闭中断来实现锁的唯一性是低效且不合理的，这里我们实现了自旋锁。通过原子性的操作实现加锁的途中不会被打断。`__sync_lock_test_and_set(&lk->locked,1)`，该指令原子性的将 `locked` 赋值 1 返回原先的值。

## 4.3.8 内核态切换

从该系统实现的开始到现在，该系统一直处于 **Machine** 模式下，显然这是不安全的，在 **Machine** 模式下，所有的寄存器用户都可以肆意访问，存在巨大的安全隐患。现在，我们要让系统常态在 **User** 模式，此模式下，所有比较敏感的寄存器都不可以访问，只有通过系统调用（`syscall`）让内核切换到 **Machine** 模式，让内核完成一些敏感的操作而不是让用户直接进行敏感操作。

### 1. 系统模式的切换

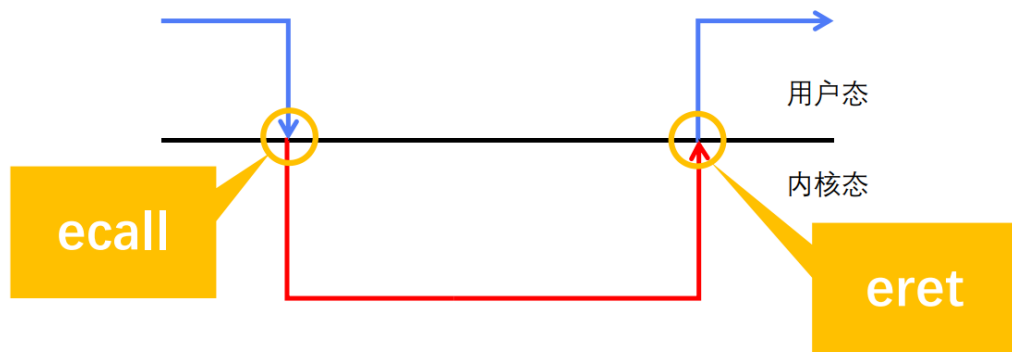


图 4-17 内核的切换

### 2. 相关指令

Call: ECALL 命令用于主动触发异常，根据调用 ECALL 的权限级别产生不同的 exception code

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

图 4-18 异常码表

### 3. 系统调用的执行流程

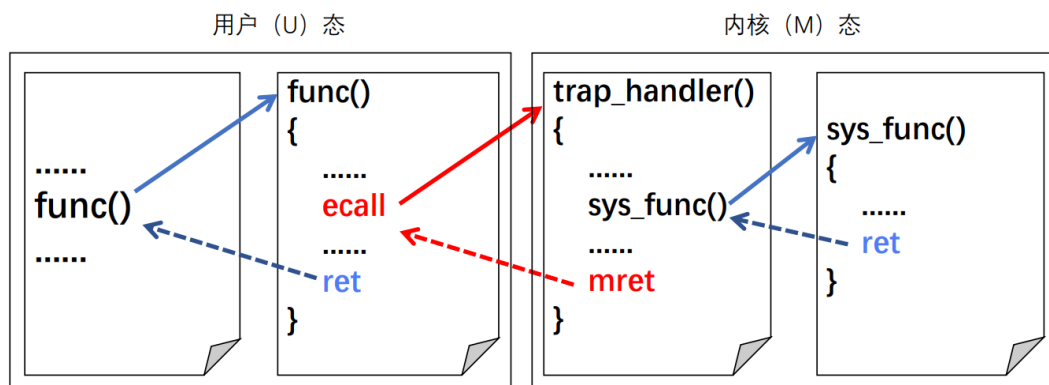


图 4-19 用户态和内核态的切换

#### 4. 系统调用的传参

系统调用作为操作系统的对外接口，由操作系统的实现负责定义。参考 Linux 的系统调用，RVOS 定义系统调用的传参规则如下：

- 系统调用号放在 a7 中
- 系统调用参数使用 a0 ~ a5
- 返回值使用 a



## 5.编码实现

### 5.1 引导和系统加入

#### 1. 进入系统

```
_start:
    csrr    t0,mhartid
    mv     tp,t0
    bnez    t0,park

    slli    t0,t0,10           # 左移 10 位
    la     sp,stacks + STACK_SIZE
    add    sp,sp,t0
    j      start_kernel

# 本系统暂时只运行在一个 hart 上，其他的 hart 不断休眠
park:
    wfi
    j      park

.align 16
```

1. 判断当前 hartid
2. hartid 不为 0 的 hart 跳走，为 0 的初始化栈指针
3. 本系统暂时只运行在一个 hart 上，其他的 hart 不断休眠

### 5.2 UART 串口设备

#### 1. 对 UART 串口设备进行内存映射

```
// 定义 UART0 的物理地址          uart 设备在内存上的映射
#define UART0    0x10000000L
```

定义 UART0 的物理地址为 0x10000000L

#### 2. 对 Uart 的寄存器操作定义相关的宏

```
#define UART_REG(reg) ((volatile uint8_t *) (UART0+reg))

#define RHR 0    // Receive Holding Register (read mode)
#define THR 0    // Transmit Holding Register (write mode)
#define DLL 0    // LSB of Divisor Latch (write mode)
```

```

#define IER 1    // Interrupt Enable Register (write mode)
#define DLM 1    // MSB of Divisor Latch (write mode)
#define FCR 2    // FIFO Control Register (write mode)
#define ISR 2    // Interrupt Status Register (read mode)
#define LCR 3    // Line Control Register
#define MCR 4    // Modem Control Register
#define LSR 5    // Line Status Register
#define MSR 6    // Modem Status Register
#define SPR 7    // ScratchPad Register

/
#define LSR_RX_READY  (1<<0)
#define LSR_TX_IDLE   (1<<5)

#define uart_read_reg(reg)  (*(UART_REG(reg)))           // 读某个寄存器 的
值
#define uart_write_reg(reg,v) (*(UART_REG(reg)) = (v))   // 向某个寄存器 写
入 v

```

读某个寄存器：uart\_read\_reg(reg)

写某个寄存器：uart\_write\_reg(reg,v)

### 3. 初始化 UART 设备

```

void uart_init()
{
    uart_write_reg(IER,0x00);    // 在 IER 写入 0，关闭中断，因为冲
突?????

    uint8_t lcr=uart_read_reg(LCR); //
    uart_write_reg(LCR,lcr | (1<<7));
    uart_write_reg(DLL,0x03);    // Uart 寄存器是 8 位，放一个 16 位数，需要两
个寄存器，DLL，是低位
    uart_write_reg(DLM,0x00);    // DLM 是高位

    lcr=0;                        // 设置奇偶校验位
    uart_write_reg(LCR,lcr|(3<<0));
}

```

\* 1. Uart 设备有自己的一套寄存器，各种读写操作本质上是通过写入它的这些专属寄存器实现的

- \* 2. 这些专属寄存器是 8 位的，有读模式(read mode)和写模式(write mode)。
- \* 3. 下面这些宏就是每个寄存器相对于 UART 的物理地址的 偏移量
- \* 4. 有一些功能是公用一个寄存器，所以 是相同的偏移量

为什么需要写 LCR 寄存器？

因为 DLL、DLM 和读和写寄存器 复用了地址，

// 要么使人中断的效果，要不使用设置波特率的效果，功能只能 2 选 1.

// 我们通过拨动一个开关来选择我们要的效果（设置 LCR 寄存器的第 7 位）

#### 4. 实现 IO 输出

```
int uart_putc(char ch){
    /**
    while ((uart_read_reg(LSR) & LSR_TX_IDLE)==0);

    return uart_write_reg(THR,ch);
}

void uart_puts(char *s){
    while (*s)
    {
        while (*s)
        {
            uart_putc(*s++);
        }
    }
}
```

不断的读 LSR 寄存器的值，同时 与操作不断这个寄存器第 5 位是否为 1，不满足条件说明 发送寄存器不空闲，如果满足条件，就把一个字符写入 发送寄存器。

## 5.3 内存管理

### 1. 申请一个专门的内存空间交给内核管理

```
/* 指定输出的架构 */
OUTPUT_ARCH( "riscv" )

ENTRY( _start )
MEMORY
{
    /* RAM 是随机存取存储器，ROM 是只读存储器（Read-Only Memory）*/
    ram (wxa!ri) : ORIGIN = 0x80000000, LENGTH = 128M /* 申请 起始地址
SECTIONS
{
    .text : {

        PROVIDE(_text_start = .); /* PROVIDE,相当于定义一个全局变量，.代表
当前地址，_text_start 记录这个当前地址*/
        *(.text .text.*) /* 把输入 section 的.text 放入 目标 saction
的 .test */
        PROVIDE(_text_end = .); /* 把 当前指针地址 记录为 _text_end */

    } >ram /* .text 的全部 放在 ram 这块内存 */

    .rodata : {
        PROVIDE(_rodata_start = .);
        *(.rodata .rodata.*)
        PROVIDE(_rodata_end = .);
    } >ram

    .data : {
        . = ALIGN(4096);
        PROVIDE(_data_start = .);

        *(.sdata .sdata.*)
        *(.data .data.*)
        PROVIDE(_data_end = .);

    }>ram

    .bss : {
        /*
```

```

        *
        PROVIDE(_bss_start = .);
        *(.sbss .sbss.*)
        *(.bss .bss.*)
        *(COMMON)
        PROVIDE(_bss_end = .);

    }>ram

/* 定义_memory_start 变量, 存储 名为 ram 这块内存开始的地址 */
/* 定义_memory_end 变量, 存储 名为 ram 这块内存结束的地址 */
    PROVIDE(_memory_start = ORIGIN(ram));
    PROVIDE(_memory_end = ORIGIN(ram)+LENGTH(ram));

    PROVIDE(_heap_start = _bss_end);                                /* ram 最后的节
是.bss, .bss 的结束地址就是 就是 heap 的开始 */
    PROVIDE(_heap_size = _memory_end - _heap_start);                /* heap 的大小
是 .bss 的结束 到 memory 的结束 的所有内存 */
}

```

## 2. 定义内存管理的宏操作

```

#include "os.h"

// 使用 mem.S 的定义的变量
extern uint32_t TEXT_START;
extern uint32_t TEXT_END;
extern uint32_t DATA_START;
extern uint32_t DATA_END;
extern uint32_t RODATA_START;
extern uint32_t RODATA_END;
extern uint32_t BSS_START;
extern uint32_t BSS_END;
extern uint32_t HEAP_START;
extern uint32_t HEAP_SIZE;

static uint32_t _alloc_start=0;
static uint32_t _alloc_end=0;
static uint32_t _num_pages=0;

#define PAGE_SIZE 4096
#define PAGE_ORDER 12    //? ? ?

#define PAGE_TAKEN (uint8_t)(1<<0) //

```

```

#define PAGE_LAST (uint8_t)(1<<1)//

/*
*页面描述符
*标志:
* -位 0:表示该页已被占用(已分配)
* -位 1:表示该页是分配的内存块的最后一页
*/
struct Page
{
    uint8_t flags;// 8 位
};

static inline void _clear(struct Page *page){
    page->flags=0;
};

static inline int _is_free(struct Page *page)
{
    if(page->flags & PAGE_TAKEN){
        return 0;
    }else{
        return 1;
    }
}

static inline void _set_flag(struct Page *page,uint8_t flags)
{
    page->flags |=flags;
};

static inline int _is_last(struct Page *page)
{
    if(page->flags & PAGE_LAST){
        return 1;
    }else{
        return 0;
    }
}

// ??????????????????
static inline uint32_t _align_page(uint32_t address)//将地址与页面边框对齐
(4K)
{

```

```
uint32_t order=(1<< PAGE_ORDER)-1;
return (address+order) & (~order); // ~ 按位取反运算符
}
```

\* \_alloc\_start 指向堆池的实际起始地址

\* \_alloc\_end 指向堆池的实际结束地址

\* \_num\_pages 保存我们可以分配的实际最大页面数。

page\_token:用第一位表示自己是否被用掉 ,这是一个掩码,

page\_last:当好几个 page 被化成一块给用户时,用第二位表示自己是不是 该内存块的 最后一个

#### 4. 内存管理的初始化

```
void page_init()
{
    _num_pages=(HEAP_SIZE / PAGE_SIZE)-8; //计算 我们可以分配的实际最大页面数
    printf("HEAP_START = %x, HEAP_SIZE = %x, num of pages = %d\n", HEAP_START,
HEAP_SIZE, _num_pages);

    struct Page *page=(struct Page *)HEAP_START; //定义初始化一个 页指针 指
    向 HEAP_START
    for(int i=0;i<_num_pages;i++){ // 把所有的页 全部清零
        _clear(page);
        page++;
    }

    _alloc_start=_align_page(HEAP_START+8*PAGE_SIZE); //动态堆可分配的 开
    始
    _alloc_end=_alloc_start+(PAGE_SIZE*_num_pages); //动态堆可分配的 结束

    printf("TEXT: 0x%x -> 0x%x\n", TEXT_START, TEXT_END);
    printf("RODATA: 0x%x -> 0x%x\n", RODATA_START, RODATA_END);
    printf("DATA: 0x%x -> 0x%x\n", DATA_START, DATA_END);
    printf("BSS: 0x%x -> 0x%x\n", BSS_START, BSS_END);
    printf("HEAP: 0x%x -> 0x%x\n", _alloc_start, _alloc_end);
}
```

初始化时,先将现有内存全部分页并设为空闲,在将每段的起始地址和大小打印出来。

## 内存页的申请和释放

```
void *page_alloc(int npages)
{
    int found=0;
    struct Page *page_i = (struct Page *)HEAP_START;
    for(int i = 0; i <= (_num_pages - npages); i++){
        if(!_is_free(page_i)){
            found=1;

            struct Page *page_j = page_i + 1;
            for(int j = i + 1; j < (i + npages); j++){
                if(!_is_free(page_j)){ // 该段连续的可分配空间不够，放弃，
继续重新找
                    found=0;
                    break;
                }
                page_j++;
            }

            if(found){
                struct Page *page_k=page_i;
                for (int k = i; k < (i+npages); k++)
                {
                    _set_flag(page_k,PAGE_TAKEN);
                    page_k++;
                }
                page_k--;
                _set_flag(page_k,PAGE_LAST);
                return (void *)(_alloc_start+i*PAGE_SIZE);
            }
        }
        page_i++;
    }
    return NULL;
}

void page_free(void *p)
{
    if(!p || (uint32_t)p >= _alloc_end){
        return;
    }
    struct Page *page = (struct Page *)HEAP_START;
```



```

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

```

申请内存页时，所有内存页顺序遍历，如果发现有连续的满足申请数量的内存页就返回其内存页地址并标识这些内存页为占用。

释放内存页时，从传过来的指针开始直到遇到第一个 last 标识，把这一段内存页全部设为空闲

## 5.4 Trap 和 Exception（异常）

1. 将 Trap 处理函数的基地址设为全局变量

```
.globl trap_vector
```

2. 在 mtvec 寄存器 写入 trap 处理函数的基地址

```

void trap_init()
{
    w_mtvec((reg_t)trap_vector);
}

```

3. 在进入 Trap 状态时，保存当前上下文，再将当前上下文地址保存到 mpc 中（等 tarp 结束后通过 mret 命令通过该寄存器来恢复 tarp 上下文），恢复 Trap 的上下文，进入 trap\_vector 处理函数。Trap 结束后再恢复原先的上下文。

```

trap_vector:

# mscratch 的意义是 保存当前在 hart 上运行的 task 的上下文（context）的地址。
# 补充：临时寄存器，清零没有用。对于临时寄存器坚持“先赋值再使用”的原则！！！！
# 保存 trap 前任务的寄存器上下文
csrrw t6,mscratch,t6 # 交换 t6,mscratch
reg_save t6          # 把 trap 前任务的寄存器上下文保存到 mscratch 中的地址

# 再单独保存的 t6 寄存器
mv t5,t6             # t5 指向 当前上下文

```

```

csrr    t6,mscratch    # 读 mscratch 的值到 t6
sw      t6,120(t5)      # 以 t5 为基+120 的地址上保存 t6 的值,

# 恢复 trap 任务的上下文
        csrr    mscratch,t5 # 把 mscratch 的值再从 t5 中拿回来

# 调用 trap. C 中的 trap 处理程序
csrr    a0,mepc
csrr    a1,mcause
call    trap_handler    # 相对于 trap_handler(mempc,mcause)

# Trap_handler 将通过 a0 返回返回地址。
csrr    mepc,a0

# trap 已经处理完了, 恢复 trap 前的任务上下文
csrr    t6,mscratch
reg_restore t6

# 回到我们在 trap 之前做的事情。
mret    # mret 指令, 会将 mepc 中的值恢复到 pc 中 (实现返回的效果)

```

## 5. trap\_handler 处理函数

```

reg_t trap_handler(reg_t epc, reg_t cause)
{
    reg_t return_pc = epc;    // epc 记录着 trap 前 pc 值
    reg_t cause_code = cause & 0xffff; //获得 trap 码

    if (cause & 0x80000000) {
        /* Asynchronous trap - interrupt */
        switch (cause_code) {
            case 3:
                uart_puts("software interruption!\n");
                break;
            case 7:
                uart_puts("timer interruption!\n");
                break;
            case 11:
                uart_puts("external interruption!\n");
                external_interrupt_handler();
                break;
            default:
                uart_puts("unknown async exception!\n");
                break;
        }
    }
}

```

```

    } else {
        /* Synchronous trap - exception */
        printf("Sync exceptions!, code = %d\n", cause_code);
        panic("OOPS! What can I do!");
        //return_pc += 4;
    }

    return return_pc;
}

```

## 5.5 硬件定时器和软件定时器

### 1. 定时器的初始化

```

void timer_load(int interval)
{
    /* 每个 CPU 都有一个单独的定时器中断源。 */
    int id = r_mhartid();

    *(uint64_t*)CLINT_MTIMECMP(id) = *(uint64_t*)CLINT_MTIME + interval;
}

void timer_init()
{
    timer_load(TIMER_INTERVAL);

    /* 开启 machine-mode timer interrupts. */
    w_mie(r_mie() | MIE_MTIE);

    /* 开启 machine-mode global interrupts. */
    w_mstatus(r_mstatus() | MSTATUS_MIE);
}

```

先设置硬件定时器的一个单位的时间间隔，每隔这个时间片段就会发生一次定时器中断。

### 2. 每当系统中断处理函数发现定时器中断时，就会调用定时器中断处理函数

```

case 7:
    uart_puts("timer interruption!\n");
    timer_handler();
    break;

```

### 4. 定时器中断处理函数，系统时钟\_tick 增加，更新硬件定时器重新定时

```

void timer_handler()
{

```

```

_tick++;
printf("tick: %d\n", _tick);

timer_load(TIMER_INTERVAL);
}

```

## 5.6 抢占式多任务

### 1. 任务数组和任务栈数组

```

uint8_t __attribute__((aligned(16))) task_stack[MAX_TASKS][STACK_SIZE]; //
__attribute__((aligned(16)))用法
https://blog.csdn.net/zhizhengguan/article/details/111470648

struct context ctx_tasks[MAX_TASKS];

```

### 2. 任务的创建

```

int task_create(void (*start_routine)(void))
{
    if (_top < MAX_TASKS) {
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE]; //在
        当前的任务上下文数组，初始化一个新任务栈，
        // ctx_tasks[_top].ra = (reg_t) start_routine; //start_routine 是
        一个函数指针 已废弃！！ 协作式时 switch_to 是通过 ret 指令，是把 ra 寄存
        器的值赋给 pc
        ctx_tasks[_top].pc = (reg_t) start_routine; //start_routine 是一个
        函数指针 -----!!! 抢占式时 switch_to 是通过 mret 指令，是把 mpc
        寄存器的值赋给 pc
        _top++;
        return 0;
    } else {
        return -1;
    }
}

```

如果任务的数量不大于系统支持的最大任务数量时，从任务数组和任务栈数组中开启一个任务上下文和任务栈给任务，并将该上下文的 pc 寄存器设置为任务的开始位置的指针。

### 3. 任务调度的初始化

```

void sched_init()
{
    w_mscratch(0); //任务切换初始化时，写入 0 到 mscratch 寄存器
}

```

```

/* 开启 machine-mode 软件中断 */
w_mie(r_mie() | MIE_MSIE);
}

```

任务切换初始化时 ,写入 0 到 mscratch 寄存器, 并且开启软件中断

### 3. 任务的调度

```

void schedule()
{
    if (_top <= 0) {
        panic("Num of task should be greater than zero!");
        return;
    }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(next);
}

```

当任务的调度时, 如果没有任务, 则报错提示, 如果有则将任务数组中存在的下一个任务的上下文的基地址交给 switch\_to 的系统调用

### 4. 切换任务的上下文

```

switch_to:
    # 切换 mscratch 以指向下一个任务的上下文的基地址
    csrw    mscratch, a0
    # 设置 mepc 为下一个任务所在的 PC
    lw      a1,124(a0)      # 根据基地址把该任务的 pc 取出到 a1
    csrw    mepc,a1

    # 恢复所有通用寄存器
    # 用 t6 指向下一个任务的上下文
    mv      t6,a0
    reg_restore t6

    # 真正做上下文切换的地方
    # 注意 mret 会开启全局中断???
    # ret          # ret 时, pc 寄存器 会设置为 ra 寄存器存的地址
    mret          # mret 将 PC 设置为 mepc 的值

```

将当前任务上下文保存到内存, 恢复即将进行的任务的上下文。并把 pc 的寄存器的值设为上一次执行该任务的指令的地址。继续该任务。

### 5. 抢占式的实现

```

void timer_handler()
{

```

```

_tick++;
printf("tick: %d\n", _tick);

timer_load(TIMER_INTERVAL);

schedule();
}

```

当系统时钟改变时，将发生一次任务切换。

## 5.7 锁

### 1. 中断锁

本系统由于是单核系统，而锁的意义是为在并发的程序中保护共享资源被合理使用。在单核情况下，如果关掉中断，则可以保证共享资源同一时间只能被一个程序使用。

```

#include "os.h"

int spin_lock()
{
    w_mstatus(x: r_mstatus() & ~MSTATUS_MIE); // 1111111011, 写0, 关闭中断
    return 0;
}

int spin_unlock()
{
    w_mstatus(x: r_mstatus() | MSTATUS_MIE); // 0000000100 写1, 开启中断
    return 0;
}

```

图 4-16 中断锁

### 2. 自旋锁

```

void initlock(struct spinlock *lk){
    lk->locked=0;
}

void _spin_lock(struct spinlock *lk){
    while (__sync_lock_test_and_set(&lk->locked,1)!=0);
}

```

```
void _spin_unlock(struct spinlock *lk){
    lk->locked=0;
}
```

通过原子性的操作实现加锁的途中不会被打断。  
 \_\_sync\_lock\_test\_and\_set(&lk->locked,1)，该指令原子性的将 locked 赋值 1 返回原先的值。

## 5.8 内核态切换

### 1.发出系统调用

```
ret =gethid(&hid);
```

2.将本次系统调用的调用号写入参数寄存器 a7,再执行 ecall 指令,发出软件异常,此时进入由 U 模式进入 M 模式

```
#define SYS_gethid 1
```

```
.global gethid
gethid:
    li a7,SYS_gethid    # 规定 a7 放系统调用号
    ecall
    ret
```

### 3. 系统捕获异常交给异常处理器 trap\_handler 处理

```
printf("Sync exceptions!, code = %d\n", cause_code);
switch (cause_code)
{
    case 8:
        printf("System call from U-mode\n");
        do_syscall(cxt);
        return_pc+=4;
        break;

    default:
        panic("OOPS! What can I do!");
        //return_pc += 4;
        break;
}
```

这里 trap\_handler 处理发现这是给系统调用,现在系统调用函数 do\_syscall(cxt)

4. 系统调用函数根据上下文的参数寄存器存放的调用号，执行相应调用  
sys\_gethid

```
void do_syscall(struct context *cxt)
{
    uint32_t syscall_num=cxt->a7;

    switch (syscall_num)
    {
        case SYS_gethid:
            cxt->a0=sys_gethid((unsigned int *)(cxt->a0));
            break;

        default:
            printf("Unknown syscall no: %d\n", syscall_num);
            cxt->a0 = -1;
            break;
    }

    return;
}
```

5. 这就是最终的系统调用函数，此时是 M 模式，所以可以敏感操作。

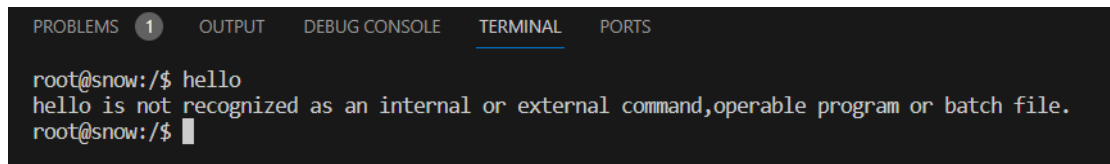
```
int sys_gethid(unsigned int *ptr_hid)
{
    printf("--> sys_gethid,arg0 = 0x%x\n",ptr_hid);
    if(ptr_hid==NULL){
        return -1;
    }else
    {
        *ptr_hid=r_mhartid();
        return 0;
    }
}
```

6. 以上执行完后，在 ret 指令下，从 M 模式恢复 U 模式。



## 6.测试

### 6.1 系统 IO 测试

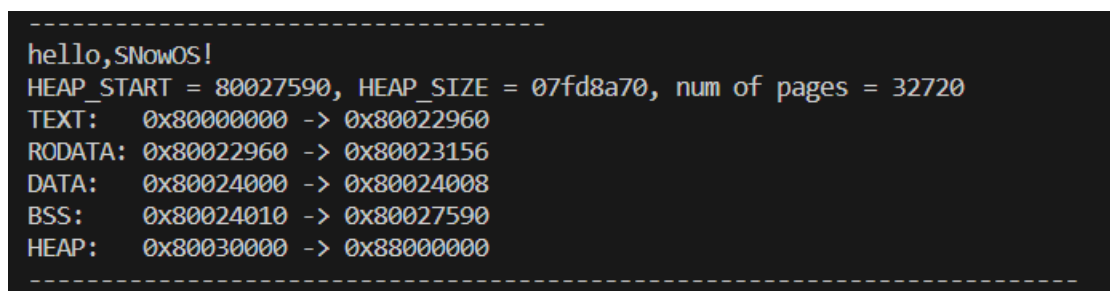
A terminal window with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is active. The prompt is root@snow:/\$. The user enters 'hello'. The output is 'hello is not recognized as an internal or external command, operable program or batch file.' followed by a new prompt root@snow:/\$.

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
root@snow:/$ hello
hello is not recognized as an internal or external command,operable program or batch file.
root@snow:/$
```

图 6-1 系统 IO 测试

输入 hello，测试显示 hello 既不是外部命令也不是内部命令。

### 6.2 内存管理测试

A terminal window showing memory allocation details. The output is enclosed in dashed lines. It starts with 'hello,SNOWOS!'. Then it shows 'HEAP\_START = 80027590, HEAP\_SIZE = 07fd8a70, num of pages = 32720'. Below this are several lines of memory layout information: TEXT, RODATA, DATA, BSS, and HEAP, each with a range of addresses.

```
-----
hello,SNOWOS!
HEAP_START = 80027590, HEAP_SIZE = 07fd8a70, num of pages = 32720
TEXT: 0x80000000 -> 0x80022960
RODATA: 0x80022960 -> 0x80023156
DATA: 0x80024000 -> 0x80024008
BSS: 0x80024010 -> 0x80027590
HEAP: 0x80030000 -> 0x88000000
-----
```

图 6-2 内存管理测试

开机输出系统的整个内存分配情况，测试显示，系统堆的起始地址是 80027590，堆的大小是 07fd8a70,整个内存有 32720 页。

### 6.3Trap 和异常测试

#### 1. 外部中断测试

```

Task 0: Running...
external interruption!
d
Task 1: Running...
Task 0: Running...
external interruption!
a
external interruption!
s
external interruption!
d
Task 1: Running...
Task 0: Running...
external interruption!
a
external interruption!
s
Task 1: Running...
external interruption!
d
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
QEMU: Terminated

```

图 6-3 外部中断测试

当对系统输入字符时，会产生一个外部中断，测试显示，每次输入确实产生了一个外部中断。

## 2. 定时器中断测试

```

Task 0: Running...
Task 1: Running...
timer interruption!
tick: 1
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
timer interruption!
tick: 2

```

图 6-4 定时器中断测试结果

这里程序设计，每当系统时钟改变 1tick 时（1 秒），会发出一个定时器中断。测试结果表明，tick 改变时，系统接受了一个定时器中断，并打印“timer interruption”。

## 3. 自定义异常

```

root@snow:/$ ptinyl
ptinyl is not recognized as an internal or external command,operable program or batch file.
root@snow:/$ Sync exceptions!, code = 5
panic: OOPS! What can I do!
QEMU: Terminated

```

图 6-5 自定义异常测试结果

测试程序写了一个异常，开机后 3 秒后触发整个异常。测试结果表明系统确实产生了整个异常码为 5 的异常，并将该异常打印出来。

## 6.4 定时器测试

### 1. 输出当前系统时间

```
root@snow:/$ time
The time is: 2023-10-10 0:7:41
root@snow:/$
```

图 6-6 输出当前系统时间的测试结果

测试方法：

当输入 time 时，系统将当前时间打印出来。

测试结果：

当前时间确实打印出来。

### 2. 检测定时器每秒发出中断

```
Task 0: Created!
timer interruption!
tick: 1
timer interruption!
tick: 2
timer interruption!
tick: 3
timer interruption!
tick: 4
timer interruption!
tick: 5
```

图 6-7 检测定时器每秒发出中断的测试结果

测试方法：

设置任务系统每秒发出中断。

测试结果：

确实每过 1s，系统就打印接受到的中断。

## 6.5 多任务测试

### 1. 开启抢占式多任务

```
root@snow:/$ time
The time is: 2023-10-10 0:7:41
root@snow:/$ system single_task_mode off
root@snow:/$ Task 1: Running...
Task 1: Running...
Task 1: Running...
```

图 6-8 开启抢占式多任务的测试结果

测试方法:

1. 输入 “system single\_task\_mode off” 关闭单任务模式。
2. 后台挂起的 Task1 恢复运行。使得我们的命令行任务和 Task1 任务并行。

测试结果:

当输入 “system single\_task\_mode off” 关闭单任务模式后，确实 2 个任务并行。根据时间片不停切换并发执行。

## 2. 创建新任务 2

```
Task 1: Running...
Task 1: Running...
Task 1: Running...
create task0
root@snow:/$ Task 1: Running...
Task 0:Created!
Sync exceptions!, code = 8
System call from U-mode
--> sys_gethid,arg0 = 0x80025478
system call returned!, hart id is 0
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
```

图 6-9 创建新任务测试结果

测试方法:

在测试前，系统只有 Task1 任务和命令行任务 2 个任务在执行。

当输入 “create task0” 时，系统将会按照命令创建新任务 Task0。

测试结果:

测试成功，在输入 “create task0” 后，出现了第 3 个任务在不断打印自己的运行情况。

### 3. 进入单任务模式测试

```
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
system single_task_mode on
Task 0: Running...
root@snow:/$ ls
ls is not recognized as an internal or external command, operable program or batch file.
root@snow:/$
```

图 6-10 进入单任务模式测试结果

测试方法:

在测试前,系统有 Task1 任务和命令行任务和 Task0 任务这 3 个任务在执行。

输入“system single\_task\_mode on”命令,开启单任务模式。

测试结果:

测试成功,在输入“system single\_task\_mode on”后,之前的 Task1 任务和 Task0 任务不在打印他们的情况,只剩下命令行任务在运行等待新的命令。

## 6.6 锁测试

### 1. 关闭锁

```
root@snow:/$ system single_task_mode off
root@snow:/$ Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
Task 1: Running...
Task 0: Running...
```

图 6-11 关闭锁结果

单任务模式是依靠上锁实现的,系统存在一个全局锁,当这个锁关闭时,所有任务都可以正常并发执行。

### 2. 开启锁

```

system single_task_mode on
root@snow:/$ ls
ls is not recognized as an internal or external command,operable program or batch file.
root@snow:/$ ls
ls is not recognized as an internal or external command,operable program or batch file.
root@snow:/$ ls
ls is not recognized as an internal or external command,operable program or batch file.
root@snow:/$ ls
ls is not recognized as an internal or external command,operable program or batch file.
root@snow:/$

```

图 6-12 开启锁结果

当这个全局锁上锁后，现在只有命令行任务获得该锁，其他任务无法发现无法获得锁，就无法被切换。

## 6.7 内核态的切换和系统调用

```

hello,SNowOS!
HEAP_START = 80026190, HEAP_SIZE = 07fd9e70, num of pages = 32721
TEXT: 0x80000000 -> 0x80021d88
RODATA: 0x80021d88 -> 0x80022168
DATA: 0x80023000 -> 0x80023004
BSS: 0x80023010 -> 0x80026190
HEAP: 0x8002f000 -> 0x88000000
Task 0:Created!
Sync exceptions!, code = 8
System call from U-mode
--> sys_gettid,arg0 = 0x80023878
system call returned!, hart id is 0

```

图 6-13 内核态的切换的测试结果

测试过程：

系统开始运行时处于 U 模式（也就是 User 模式），该模式下，很多设计敏感操作的寄存器无法被访问。

我们这里想要获取当前运行的 hart 的 id，然而该行为属于敏感操作，不能在 U 模式下执行，此时我们通过系统调用 `gettid()`，让内核帮我们获取 hart 的 id。

系统调用 `gettid()` 时，内核将切换成 M 模式，并执行内核本身的相关函数，获取 Hart 的 id，系统调用结束后，恢复从 U 模式并不结果传递给上层调用者。

测试结果：

测试成功，这里使用系统调用后，打印出此时从 U 陷入 M 模式，并打印出 hart id 是 0。

## 7.项目总结

在本项目中，我使用 C 和汇编语言成功地开发了一个基于 RISC-V 指令集架构的 32 位支持单核的操作系统。该操作系统经过一系列的功能实现，包括引导和系统入口、UART 串口设备、内存管理、上下文切换和协作式多任务、Trap 和 Exception、外部设备中断、硬件定时器、抢占式多任务、任务同步和锁、软件定时器、内核态切换，并支持简单命令行。

在整个开发过程中，我遇到了一些挑战和困难。特别是在中断和异常模块的编写过程中，经常遇到错误和异常，需要耐心地进行调试和排查。然而，最终我成功地解决了这些问题，并为操作系统增加了强大的功能。

最让我感到自豪的是命令行交互的实现。通过命令行，我能够与操作系统进行实时的交互和控制。我可以使用命令获取系统时间，通过命令在后台开启新的任务，通过 `clear` 命令清理屏幕上的输出。最重要的是，我可以通过命令在单任务和多任务之间实时切换，充分利用系统的资源和提高效率。这让我真正感受到这个操作系统不再是一个简单的玩具，而是一个真正功能完备的操作系统。

通过这个项目，我不仅学习了操作系统的原理和开发技术，还增强了我的编程能力和问题解决能力。我意识到了操作系统开发的复杂性和挑战性，同时也体验了解决问题带来的成就感和满足感。

未来，我将继续改进和完善这个操作系统，修复中断和异常模块的问题，并进一步优化和拓展其他功能模块。我希望通过持续的努力和学习，将这个操作系统打造成一个更加稳定、高效和功能丰富的系统，为使用它的用户提供更好的使用体验。