**操作系统及安全课程设计实验报告**

实验题目： 使用C语言开发一个基于

RISC-V架构的操作系统

组 号：

指导教师：

小组成员姓名学号：

**张三(16222222)**

**李四(16111111)**

**王五(16222222)**

**赵六(16333333)**

日 期： 年 月 日

目 录

[一 实验介绍 1](#_Toc194854174)

[二 实验任务描述 1](#_Toc194854175)

[三 实验目的 1](#_Toc194854176)

[四 实验任务 1](#_Toc194854177)

[1. 实验1：实验环境概要 1](#_Toc194854178)

[1.1 实验描述 1](#_Toc194854179)

[1.2 实验步骤 2](#_Toc194854180)

[1.3 实验步骤 2](#_Toc194854181)

[三 遇到问题及解决方法 2](#_Toc194854182)

[四 实验过程、核心代码及实验结果展示 2](#_Toc194854183)

[五 个人实验改进与总结 2](#_Toc194854184)

[5.1 个人实验改进 2](#_Toc194854185)

[5.2 个人实验总结 3](#_Toc194854186)

[六 参考文献 3](#_Toc194854187)

(大家注意，目录是自动生成的，页码从正文部分开始，当同学们把正文写完后，只需要右击目录，选择更新域，目录会自动更新)

# 一 实验介绍

本实验以RISC-V架构上的操作系统[TATAK OS](https://github.com/yztz/tatakOS)为基础，结合操作系统及安全的理论课，设置了环境配置、运行与调试、系统调用、进程管理、内存管理、文件管理、实验进阶等若干个小实验，使同学能够了解在异构平台上开发RISC-V操作系统的基本概念、步骤和方法，并将相关理论知识与具体代码实现相结合，掌握代码的修改和调试方法。

# 二 实验任务描述

实验有若干个小任务组成， 采用小组协作方式进行，每小组最多4人，自设一名组长组织小组的分工合作，第一次上机时由组长将小组成员名单（包括组长姓名、成员姓名及学号）交给老师 ，每个小任务由小组内指定其中一人在头歌平台提交任务评测，但要求小组内所有学生都掌握实验的原理、方法和结果，所有实验最终撰写一份实验报告，指导教师在实验指导过程中进行实验的抽样验收。具体实验任务发布在头歌平台<https://www.educoder.net/>。

# 三 实验目的

* 掌握RISC-V异构平台开发环境配置与调试方法
* 理解操作系统核心模块在RISC-V架构下的实现
* 具备通过代码修改实现系统功能扩展的能力
* 形成从理论到实践的完整开发链路认知

# 四 实验任务

## 2.实验2：运行与调试

**主要完成人**：**刘翠桐** **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 2.1实验描述

**实验介绍：本实验以 qemu 和 gdb 为调试工具，介绍操作系统的基本调试方法。（正文部分一律用 宋体，五号字，1.25倍间距，首行缩进两格）**

**实验目的：**

**掌握 qemu 的基本用法**

**掌握 gdb 的基本用法**

**掌握 qemu 和 gdb 配合使用的方法**

**掌握调试操作系统的方法**

**实验要求：**

**使用 qemu 体验系统级模拟和用户级模拟**

**学会编写自己的用户程序**

**使用 gdb 进行调试**

**完成指定的调试任务**

### 2.2实验过程

**在user/src目录下，创建一个hello.c文件，**

文本

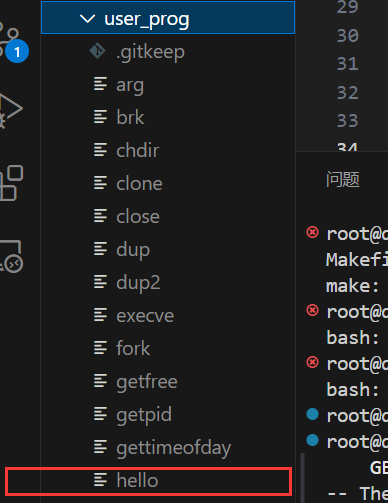
AI 生成的内容可能不正确。

图表 1创建hello.c文件

**在根目录下使用“make user”编译文件，编译完成后在build/user\_prog目录喜爱可以看到，刚刚的hello生成的可执行文件hello**

文本

AI 生成的内容可能不正确。



**启动qemu用户模式运行程序**

图形用户界面, 文本

AI 生成的内容可能不正确。

**查看系统调用日志**

文本

AI 生成的内容可能不正确。

**运行自制OS，并执行 hello ，程序发现巨多的系统调用**

文本

AI 生成的内容可能不正确。

**退出后，找到syscall函数，并修改debug过滤条件，仅当函数名为hello时输出，修改后在次在os中运行hello**

**看到如下输出**

表格

AI 生成的内容可能不正确。

### 2.3核心代码

在user/src目录下，创建一个hello.c文件，文本

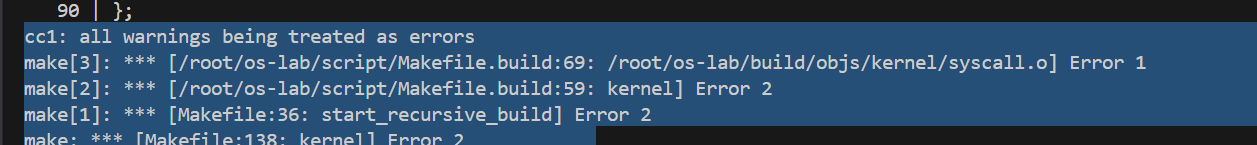
AI 生成的内容可能不正确。

**（此处详细记录修改文件的路径、文件行号和修改内容，添加必要的注释）**

### 2.4遇到问题及解决方法

**（此处详细记录修改文件的路径、文件行号和修改内容，添加必要的注释）**

**问题：在添加过滤条件之后，编译出现错误**



**解决：使用kprint函数来输出**

## 5.实验5：系统调用综合

**主要完成人**： **刘翠桐**

**其他参与人员： （根据需要选填写）**

### 5.1实验描述

在系统调用综合应用中，学生需开发一个监控进程行为的sys\_trace。具体功能为监控当前进程接下来的系统调用，若发生`syscall\_num`对应的系统调用，则打印输出此次系统调用的相关信息。此外，该进程的子进程也将会继承此监听项。

### 5.2实验过程

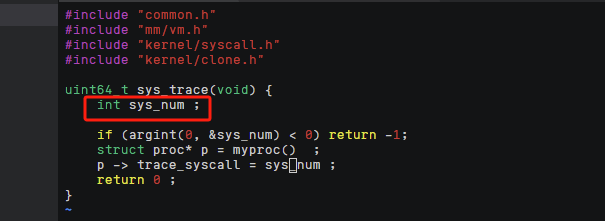
首先需要在entry/syscall.tbl文件中添加该系统调用的相关信息表项，规定系统调用号为524，具体如下图。

文本

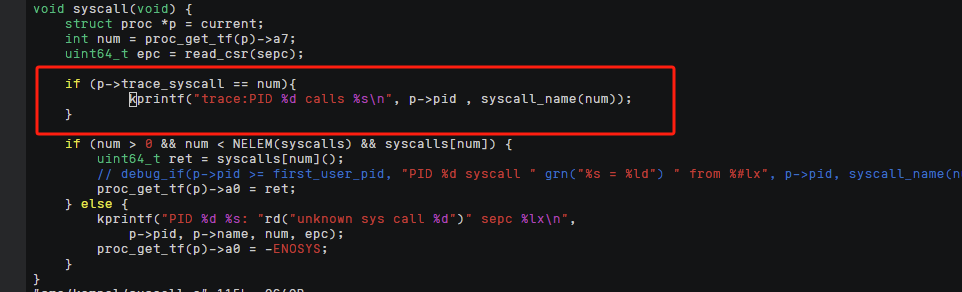
AI 生成的内容可能不正确。

接下来我们需要再src目录下建立一个sys\_trace.c用于定义该系统调用，代码如下图，参数sys\_num为需要监听的系统调用号。trace需要一个参数（系统调用号）来表示所监听的系统进程。注意：系统调用的参数传递与普通过程调用参数传递方式不太一致，需要通过特殊方法获取。事实上，正常函数参数可以从寄存器或栈中获取，而系统调用的参数则要从陷入时保存的上下文中获取。

因此我们需要调用syscall.h中的argint函数来从用户空间获取参数，其次设置一个存在于PCB结构体中的参数为trace\_syscall（具体操作后续给出）,表示所要监听的系统调用号，调用 myproc() 获取当前的 proc 结构体指针，令p -> trace\_syscall，表示当前进程所要监听的系统调用号。

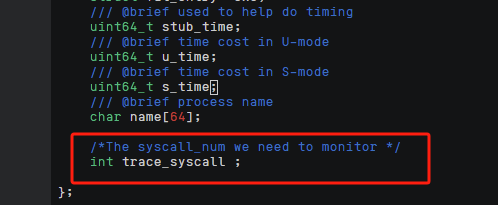


由于每当进行系统调用，如果是被监视的进程，实验要求都要打印进程号，以及调用的syscall\_name。因此直接将kprintf放入syscall(void)函数中，在syscall函数中进行判断（函数在src/kernel/syscall.c），在RISC-V中，用户空间通过a7寄存器传递系统调用号，使用num接受并于p->trace\_syscall进行比较，如果相同则打印该系统调用的相关信息。

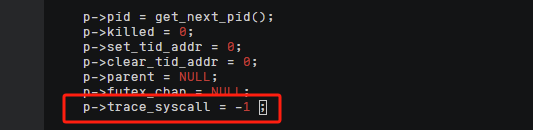


在syscall(void)函数中加入一段判断是否打印的代码，若当前进程调用的系统调用号为我们设置监听的系统调用，那么我们打印它，具体实现如上图所示。

回到刚开始时设置的参数trace\_syscall，由于这个参数是自己所设置，因此需要在PCB结构体中添加trace\_syacall的，因此我们需要修改PCB结构体：



在PCB被创建(初始化)时，我们也应该对trace\_syscall初始化，由于实验要求，当输入的系统调用号为-1时表示取消监听，因此我们设置初始化时为-1表示未启动监听。

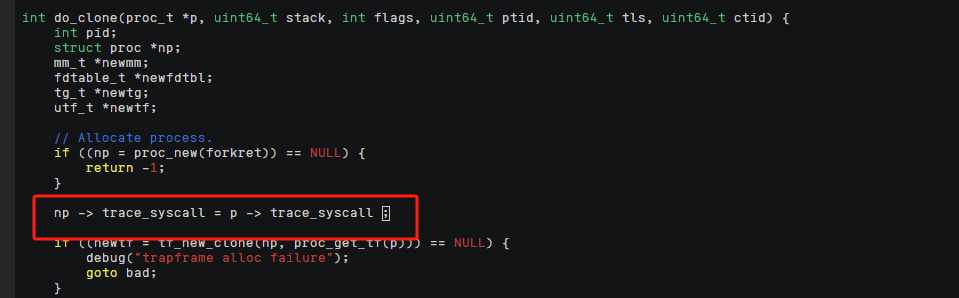


最后，为了实现父进程子进程时，子进程的PCB中trace\_syscall继承了父进程的trace\_syscall，我们需要修改fork()的代码。我使用vscode找到fork的实现函数，发现具体实现是通过do\_clone函数来实现，因此我们直接修改do\_clone()即可

文本

AI 生成的内容可能不正确。

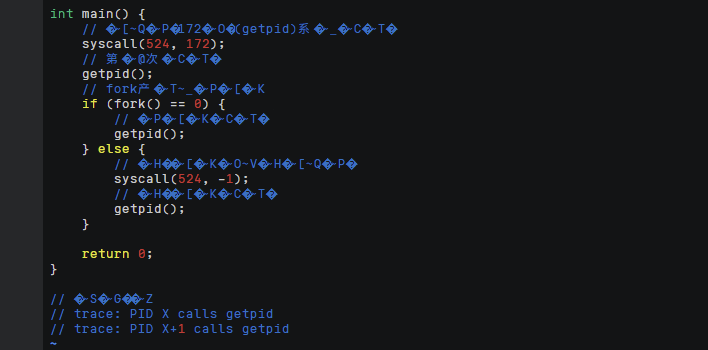
在其中添加np -> trace\_syscall = p->trace\_syscall，同步父子进程的被监视号。



编写测试代码：

在user/src目录下编写用户态下的调用代码（test\_trace.c）：

这段代码主要是监听172号系统调用（getpid()）

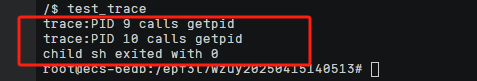


编译后运行测试

编译并启动qemu虚拟机：

./run-qemu.sh

在qemu提供的命令行执行test\_trace，可以发现此时能够成功的监听到172号的系统调用以及监听其子进程。

发现调用成功了！

### 5.3核心代码

系统调用实现：

uint64\_t sys\_trace(void) {

int sys\_num ;//设置参数来接受所要监听的系统调用号

if (argint(0, &sys\_num) < 0) return -1;

struct proc\* p = myproc() ;//获取当前进程的 proc 结构体指针

p -> trace\_syscall = sys\_num ;//赋值当前进程的监听号

return 0 ;

}

向syscall.c中添加：

if (p->trace\_syscall == num){

//如果当前进程监听的系统调用号与num（当前系统调用号）符合

kprintf("trace:PID %d calls %s\n", p->pid , syscall\_name(num));

}

在proc.h中的PCB结构中添加：

int trace\_syscall ;

向proc.c中添加：

p->trace\_syscall = -1 ;//初始化trace\_syscall，默认不监听

向clone.c中添加：

//用于子进程继承父进程的监听项

np -> trace\_syscall = p -> trace\_syscall ;

测试代码(test\_trace.c)：

int main() {

syscall(524, 172);

getpid();

if (fork() == 0) {

getpid();

} else {

syscall(524, -1);

getpid();

}

return 0;

}

### 5.4遇到问题及解决方法

1. 如何获取系统调用的参数：通过上一小节（系统调用入门）的学习，采用argint函数来获取系统调用的第一个参数。

## 实验8：程序加载

**主要完成人**： **刘翠桐**

**其他参与人员： （根据需要选填写）**

### 8.1实验描述

一、实验介绍

我们在linux中可以从文件系统中加载可执行文件并运行，例如a.out：

$ ./a.out

这个实验你将亲自动手解析ELF格式可执行文件，了解用户程序是如何被操作系统加载执行，以及在整个过程中所牵涉到的关键OS功能模块。

二、实验目的

通过本次实验，你将会学习到：执行一个可执行文件的基本流程。

简单的静态链接ELF文件解析过程。执行程序所牵涉到的OS功能模块的基本概念。

三、实验要求

掌握ELF的基本结构，理解ELF文件的两种视图。理解程序的加载过程和生命周期。

### 8.2实验过程

编译内核之后，使用readelf命令查看内核ELF,

文本

AI 生成的内容可能不正确。

补全代码

文本

AI 生成的内容可能不正确。

1. **遍历Program Headers：**循环处理每个Program Header，仅处理类型为PT\_LOAD的段。
2. **转换权限标志：** 将ELF段中的权限标志（PF\_R, PF\_W, PF\_X）转换为系统内存保护标志（PROT\_READ, PROT\_WRITE, PROT\_EXEC）。
3. **映射内存区域：** 使用mmap\_map为当前段分配虚拟内存区域，指定起始地址、大小和权限。
4. **加载段内容：** 调用loadseg函数将ELF文件中的段内容加载到已映射的内存中。
5. **清零.bss段：** 如果段的内存大小（memsz）大于文件大小（filesz），则剩余部分需填充零。

**测评后通过**

图片包含 文本

AI 生成的内容可能不正确。

### 8.3核心代码

文本

AI 生成的内容可能不正确。

### 8.4遇到问题及解决方法

## 实验11 内核线程

**主要完成人**： **刘翠桐**

**其他参与人员： （根据需要选填写）**

### 11.1实验描述

**实验介绍**：在内核中，除了通常的进程外，还存在着一些“进程”，它们始终在内核态下工作，帮助内核完成一些运行时的子任务，比如，磁盘缓存脏块/脏页的定时写回等，我们称这样的“进程”为内核线程。在本实验中，我们的文件镜像中存在一个名为password的文件，你需要在此内核线程中读取此文件并输出。

**实验目的：**理解和掌握进程概念，学会使用内核线程，理解它的创建与执行过程

**实验要求：**创建指定的线程工作函数

具体步骤如下：

1. 使用kthread\_create在mythread\_init中初始化我们的线程
2. 使用namee读取password所对应的文件目录项
3. 使用reade读取文件内容
4. 使用kprintf输出内容

**知识点：**

1.namee 调用链：文件路径解析

entry\_t \*namee(entry\_t \*from, const char \*path);

from：起始目录条目（NULL表示根目录；path：目标路径（如"/password"）

2.reade 调用链：文件数据读取

int reade(entry\_t \*entry, int user, uint64\_t buf, off\_t off, int n);

entry：namee返回的文件条目；user：0表示内核缓冲区，1表示用户空间；buf：目标缓冲区（物理地址）；off：文件偏移量；n：请求读取的字节数

### 11.2实验过程

创建一个新的内核线程password\_reader线程，与当前线程并发运行，并将线程与mythread\_fn函数绑定，该线程启动后会立即执行此函数

文本

AI 生成的内容可能不正确。

实验已给出线程初始化过程：

文本

AI 生成的内容可能不正确。

后应执行：（具体实现如图）

entry\_t \*e = namee(...); // 查找文件，读取password所对应的文件目录项

reade(...); // 读取文件内容

kprintf(...); // 输出密码内容

文本

AI 生成的内容可能不正确。

在entry\_t \*entry = namee(NULL, "/password");中

NULL表示从根目录开始查找，"/password"为查找的目标路径

在int n = reade(entry, 0, (uint64\_t)buf, 0, sizeof(buf)-1);中

entry为指定操作的目标文件，0表示内核空间（1表示用户空间），(uint64\_t)buf为缓冲区地址，用于强制类型转换匹配项目API定义，sizeof(buf)-1为读取的长度，预留1字节给字符串终止符。

### 11.3核心代码

src/mythread.c文件完整内容：

#include "common.h"

#include "fs/fs.h"

#include "fs/file.h" //提供reade()声明

#include "kernel/proc.h" //提供kthread\_create()声明

#include "driver/console.h"

extern fat32\_t \*fat;

void mythread\_fn(proc\_t \*me) {

// 等待文件系统初始化

while (!fat)

yield();

\_\_sync\_synchronize();

// 1. 使用项目API查找文件

entry\_t \*entry = namee(NULL, "/password");

if (!entry) {

kprintf("Failed to find /password\n");

while(1);

}//失败处理

// 2. 使用项目API读取文件

char buf[256];

int n = reade(entry, 0, (uint64\_t)buf, 0, sizeof(buf)-1);

if (n <= 0) {

kprintf("Failed to read /password\n");

eput(entry);

while(1);

}//失败处理

buf[n] = '\0';

// 3. 输出纯净内容（无额外格式）

kprintf("%s", buf);

// 4. 释放资源

eput(entry);

while (1);

}

void mythread\_init() {

if (kthread\_create("password\_reader", mythread\_fn) < 0) {

kprintf("Failed to create thread\n");

}

}

### 11.4遇到问题及解决方法

## 14.实验14：FAT32簇链

**主要完成人**：**刘翠桐**  **整个实验任务**

### 14.1实验描述

我们都知道，操作系统文件的分配方式主要有链式分配、索引分配等等。而FAT32则是一种典型的显示链式文件分配方式。在本节实验中，将完成FAT32簇链的追踪。

**关键点：**

理解链式分配文件系统的簇链遍历

体会链式分配文件系统的优缺点

完成实现指定函数功能

**实验思路：**

FAT32文件系统是显示链式分配的文件系统，因此它存在一张fat表（忽略fat表的备份），里面记录了文件区块的链式信息（FAT32以簇为粒度组织文件区块），FAT32中的每一个簇在fat表中都对应着一个4字节大小的表项，表项中存储的内容即是下一个簇的簇号，因此只要不断遍历下一个簇号，即可得到文件的所有簇。需要做的是给定一个簇号，返回它的后继簇号：

*// src/fs/fat32.c*

static uint32\_t fat\_next\_cluster(fat32\_t \*fat, uint32\_t cclus) {

    if(IS\_FAT\_CLUS\_END(cclus) || cclus == FAT\_CLUS\_FREE)

        panic("try to follow a FREE/EOC cluster");

    uint32\_t answer;

    blk\_buf\_t \*buf = bread(fat->dev, clus2fatsec(fat, cclus));

    todo("We have read the relevant disk blocks for you, then you just need to query the information and return it.");

    brelse(buf);

    return answer;

}

实验已经基于给定簇号，读取出了其在fat表中的扇区内容，我们需要查询当前簇cclus对应的表项，并读取出下一个簇号，将其记录在answer变量中。

### 14.2实验过程

（1）编写代码

改代码需要总体实现以下逻辑：

检查簇号是否合法；

找到该簇在 FAT 表中的位置；

从对应扇区读取 FAT 表项；

返回该簇所指向的下一个簇号。

源代码已经给定簇号，读取出了其在fat表中的扇区内容，只需要计算当前偏移位置，再根据当前位置和偏移位置计算下一个簇号，并将其记录在answer变量中，最后可以通过打印输出调试信息来证明以上过程的正确性。

文本

AI 生成的内容可能不正确。

图 4-1 完整fat\_next\_cluster代码

（2）结果显示

以下为进入内核后的打印结果，可以看到显示了当前簇号和后续簇号，第二条显示的当前簇号正是上一条的下一个簇号，由此可见前述代码正确运行。

文本

AI 生成的内容可能不正确。

图 4-2 打印结果

### 14.3核心代码

文本

AI 生成的内容可能不正确。

图 4-3 核心代码

### 遇到问题及解决办法

## 16.实验16：时钟中断

**主要完成人**： **刘翠桐**

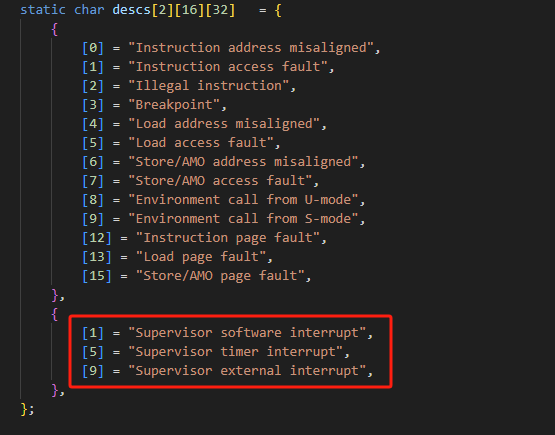
**其他参与人员：**

### 16.1实验描述

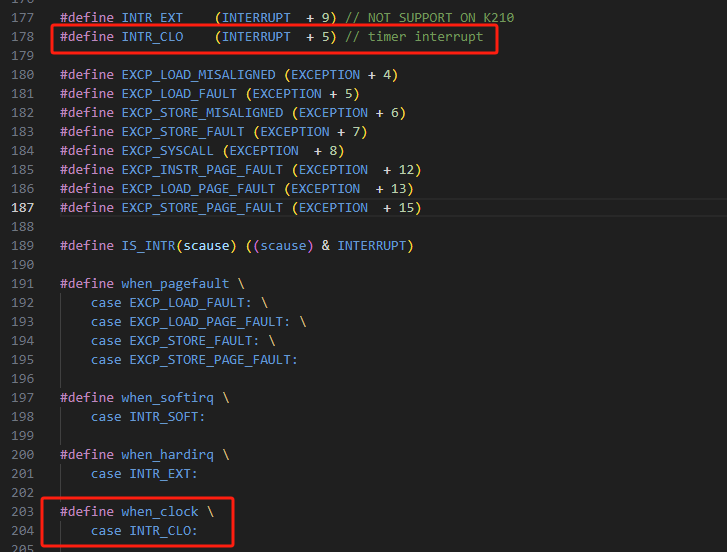
本实验通过配置时钟中断处理机制，要求学生掌握中断触发与响应的完整流程，包括初始化中断描述符表、注册中断服务例程、实现中断上下文保存与恢复等核心操作，重点理解时钟中断周期性触发特性及其在任务调度计时、超时检测等系统功能中的基础作用，通过编写中断处理函数并观察中断触发时寄存器的压栈与恢复过程，结合日志输出分析中断嵌套与屏蔽控制逻辑，最终建立对硬件中断与操作系统协同工作机制的直观认知。

### 16.2实验过程

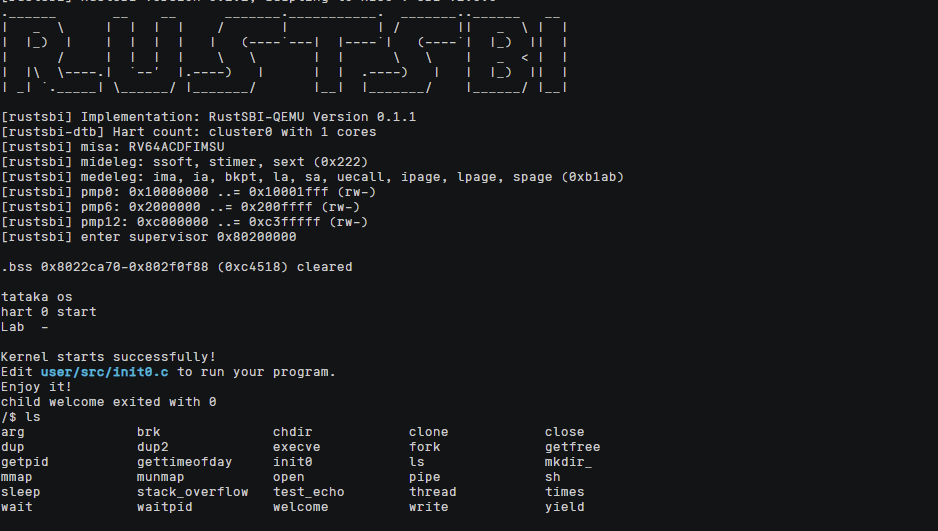
本次实验的核心就在于在中断向量表中添加一段给时钟中断进入的入口地址：



如图为中断向量表的描述，中断地址+5处即为本实验时钟中断的入口。



定义时钟终端的地址为INTERRUT+5，符合中断向量表中的地址，之后添加至when\_clock中。



再次编译运行虚拟机，就发现已经成功编译通过了！

### 16.3核心代码

文件：/include/risv.h：

+第178行： #define INTR\_CLO (INTERRUPT + 5) // timer interrupt

+第204行： case INTR\_CLO:

### 16.4遇到问题及解决方法

# 五 实验改进与总结

## 5.1 实验改进

**（此处重点突出自己如何改进，为什么这样改进，说出自己改进部分的优缺点，有效文字不少于200字）**

## 5.2 实验总结

**（小组成员每人都要填写）**

**（此处类似于写观后感，同学们辛辛苦苦完成实验，单独编译内核可能就折腾很久，相信会有很多的感触和收获，让你的笔在第一时间记录这美好的瞬间，有效文字不得少于200字）**

# 六 参考文献

**（此处罗列同学们在做实验过程中 所参考的文献、书籍、网站， 或者向某个高手同学请教都可以写， 大多数同学应该是参考 网站、书籍和问人比较多，以条目化的形式呈现， 如1. 2. 3. ）**

# 七 组内自评

|  |  |  |
| --- | --- | --- |
| **组员** | **主要完成任务内容及表现** | **评价** |
| **张三** |  | **优秀**  **良好**  **中等**  **及格** |
| **李四** |  | **优秀**  **良好**  **中等**  **及格** |
| **王五** |  | **优秀**  **良好**  **中等**  **及格** |
| **赵六** |  | **优秀**  **良好**  **中等**  **及格** |