**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

## 实验1：实验环境概要

**主要完成人： 王书利**  **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

**初步熟悉实验环境，通过对实验环境的了解，理解后续实验操作的目的和意图。**

**1. 项目的结构**

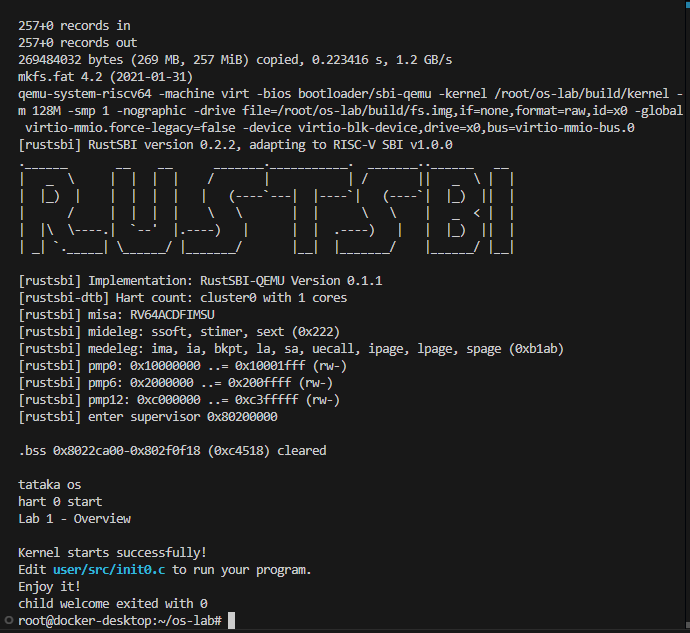
**我们目前只需要重点关注src, user以及根目录下的几个.sh脚本文件即可。**

**2. 编译与运行**

**在命令行输入：$ ./run-qemu.sh你将会看到内核与文件镜像被自动构建生成，并在QEMU中运行**

### 实验过程

**编译与运行，命令行输入./run-qemu.sh**



**图1. 内核与文件镜像构建生成**

### 核心代码

**./run-qemu.sh 在qemu中运行构建的内核**

### 遇到问题及解决方法

**不知如何进入os-lab目录，后通过互联网查询使用cd指令进入。**

## 实验2：运行与调试

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 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**

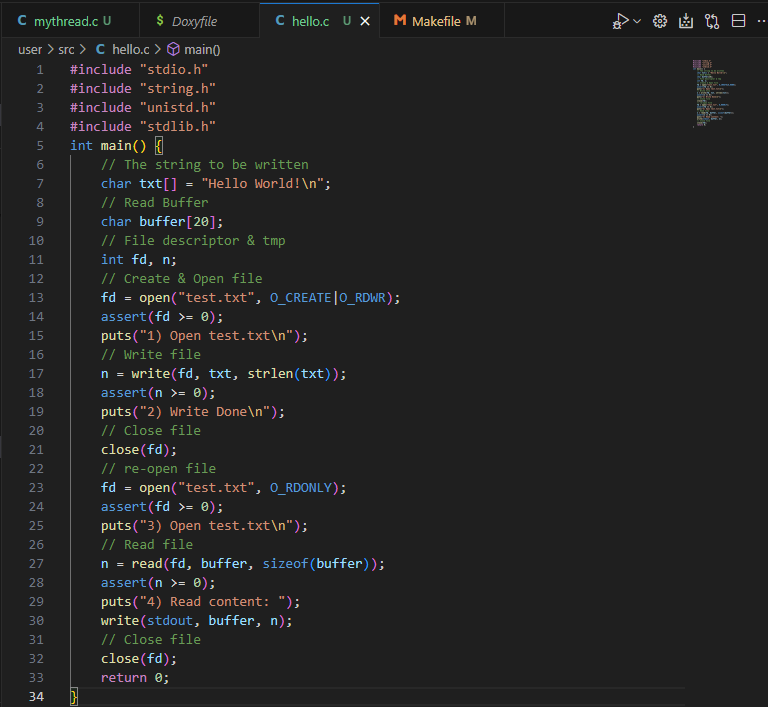
**学会编写自己的用户程序**

**使用 gdb 进行调试**

**完成指定的调试任务**

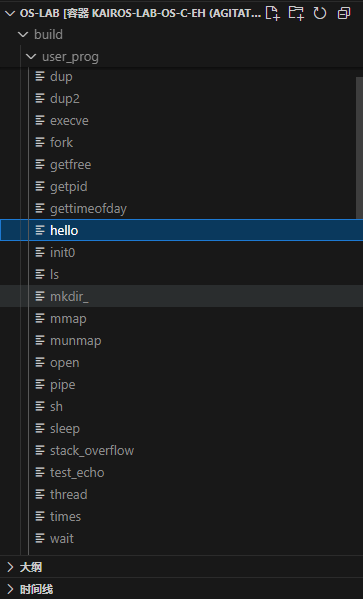
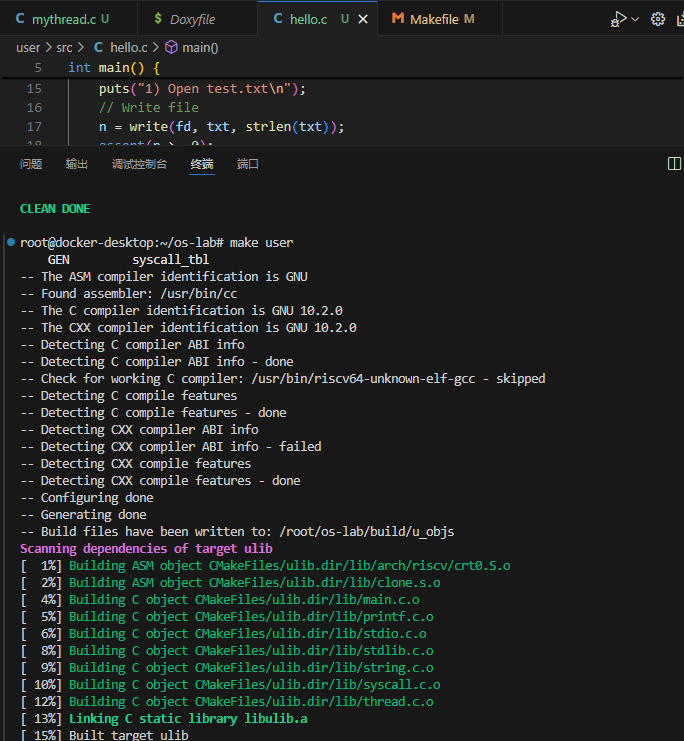
### 实验过程

**在user/src目录下，创建一个hello.c文件**

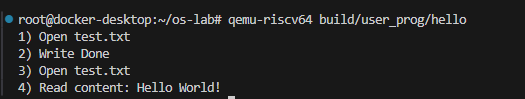


**图1. 创建hello.c文件**

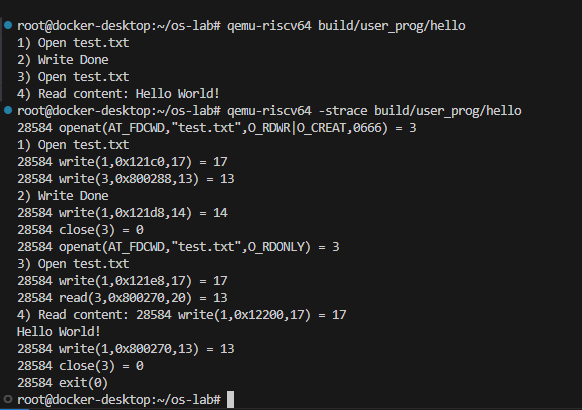
**在根目录下使用“make user”编译文件，编译完成后在build/user\_prog目录喜爱可以看到，刚刚的hello生成的可执行文件hello**



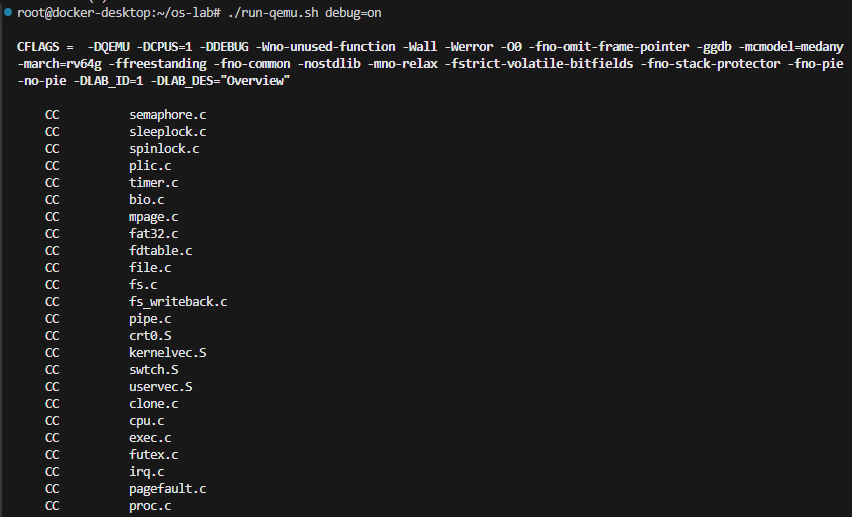
**启动qemu用户模式运行程序**



**查看系统调用日志**



**运行自制OS，并执行 hello ，程序发现很多系统调用**

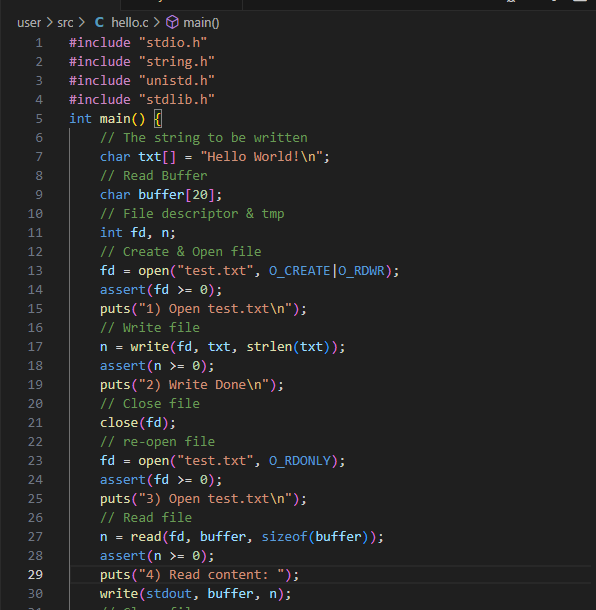


**退出后，找到syscall函数，并修改debug过滤条件，仅当函数名为hello时输出，修改后在次在os中运行hello**

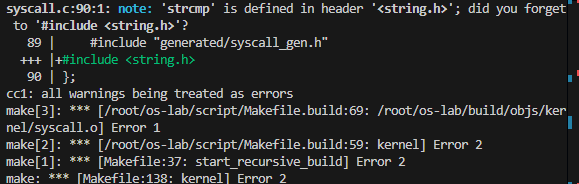
表格

AI 生成的内容可能不正确。

### 核心代码



### 遇到问题及解决方法

****

## 实验3：调试实战

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

在本实验中，你将面临一个bug，你要动用各种你所知道的各种调试手段来完成调试任务。通过实践来理解调试的复杂性，思考如何寻找到调试的最佳实践，在实战中锻炼系统调试能力。操作系统在启动后需要引导第一个用户进程init0，而基于该进程可以衍生出一个系统进程树。

进入实验后，首先输入$ ./run-qemu.sh命令来运行操作系统，查看输出，正常情况下，你应该会看到一段错误信息，请根据错误信息的提示，尝试修复这个问题。

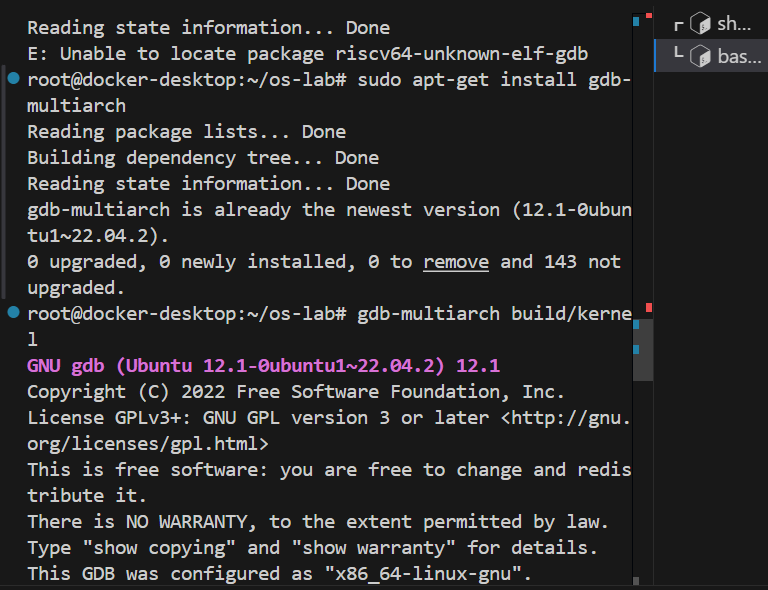
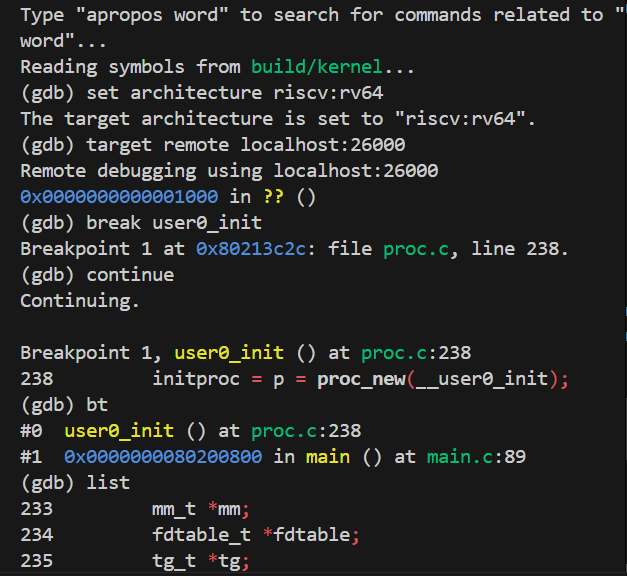
### 实验过程

1.复现报错并查看输出，运行 ./run-qemu.sh，观察启动日志，确认内核对 init0 和 welcome 的加载流程。

2.启动 GDB 远端调试端口

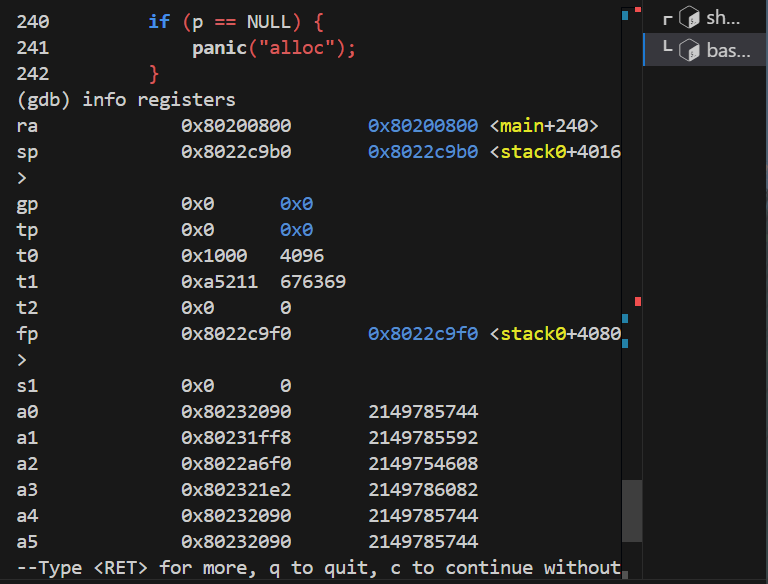
使用 make run ... EXTRA\_QEMUOPTS='-S -gdb tcp::26000' 启动 QEMU 并挂起，等待 GDB 连接。

在宿主机上通过 gdb-multiarch build/kernel，设置架构为 riscv:rv64，并执行 target remote localhost:26000。

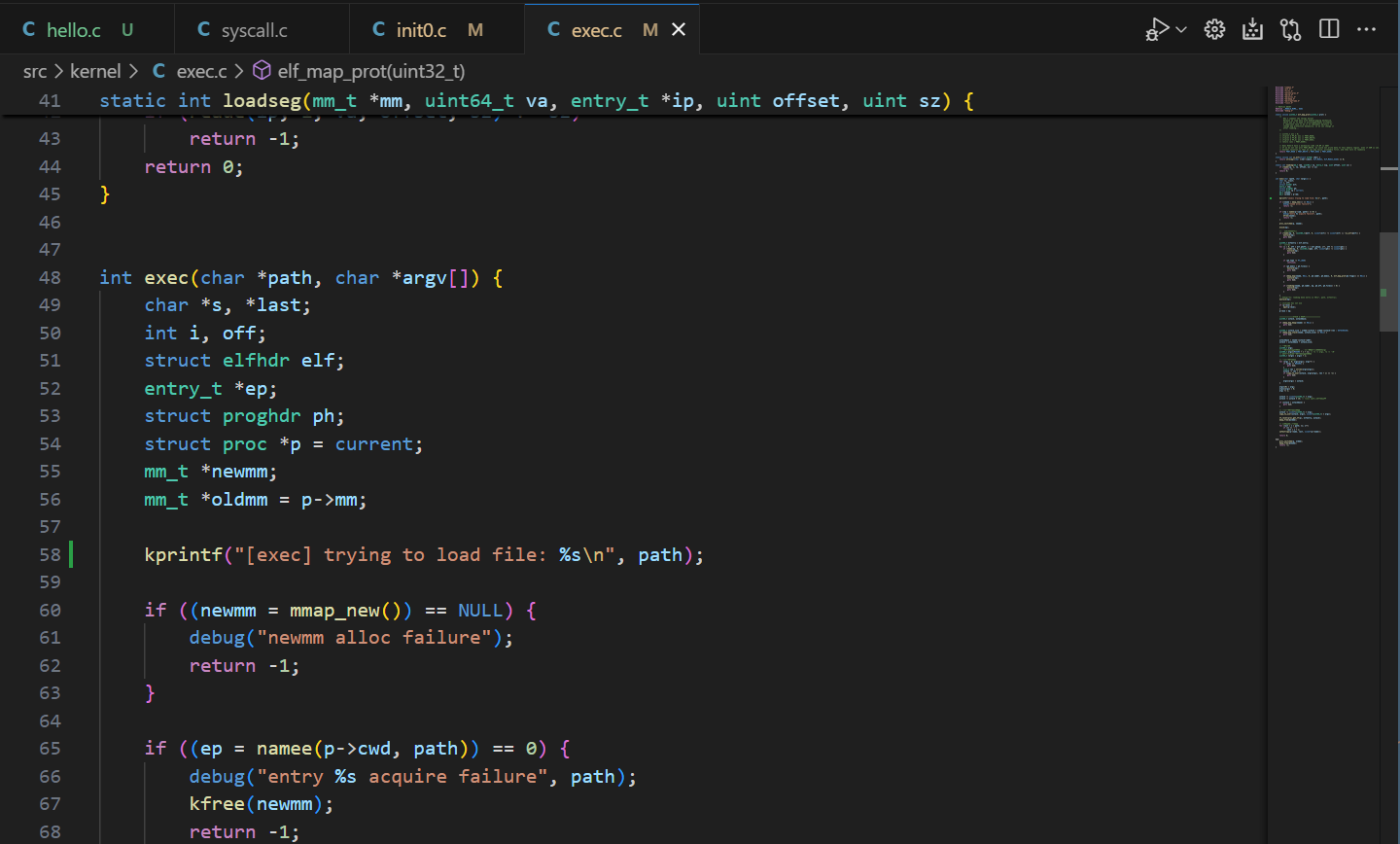
user0\_init 入口设置断点：break user0\_init continue

执行 bt、list、info registers 等命令，跟踪内核调用栈与寄存器状态，确认内核正确跳转到 proc\_new(\_\_user0\_init)



3. 打印调试 exec 路径

在 src/kernel/exec.c 的 exec(char \*path, char \*argv[]) 函数开头加入：



重新编译并运行，观察 QEMU 输出中：

[exec] trying to load file: init0

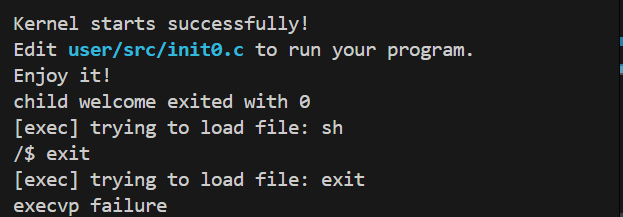
[exec] trying to load file: welcome

[exec] trying to load file: sh

确认内核按序加载 init0、welcome、sh，无 panic。

4. **用户 shell 交互流程验证**

进入交互式 shell / $，输入 exit，观察到：



shell 未对 exit 进行内建命令处理，错误地将其当成外部程序执行。

**5. 修复 shell 内建命令**

编辑 user/src/sh.c，在 fork() 之前拦截：

+ if (strcmp(argv[0], "exit") == 0) {

+ break;

+ }

重新编译并运行，在 shell 中输入 exit 时，shell 正常退出，返回 init0 并最终执行 halt() 停机。

### 核心代码

1. src/kernel/exec.c

插入debug("[exec] trying to load file: %s", path);

2. user/src/sh.c

添加 exit 内建命令- // 原始 fork/exec 逻辑

+ // 处理 exit 内建命令

+ if (strcmp(argv[0], "exit") == 0) {

+ break;

+ }

### 遇到问题及解决方法

问题：内核直接调用 namee(path)，报错信息不明确，无法知道它尝试打开哪个路径。

解决：在 exec() 函数开头插入 debug() 打印，输出每次加载的 path。

问题：在 / $ shell 中输入 exit，内核尝试加载名为 exit 的可执行文件，导致 execvp failure。

解决：在 user/src/sh.c 中的命令解析阶段，识别并拦截 exit，作为内建命令处理，直接 break 循环，正常退出 shell。

## 实验4：系统调用入门

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

本实验旨在通过实现并测试简单的系统调用函数，熟悉操作系统中系统调用的基本原理和调用流程。通过编写用户程序test\_echo，调用execve系统调用，完成用户程序的加载与执行，掌握系统调用的接口设计和内核交互方式。

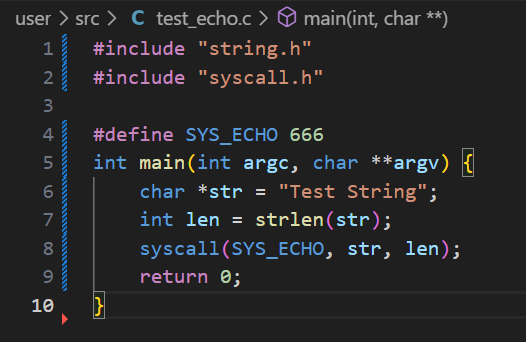
实验内容包括实现部分常用系统调用的封装，如open、close、read、write、execve等，并验证execve调用的正确性。通过本实验，考察学生对系统调用机制的理解、函数封装技巧及调试能力。

关键知识点包括系统调用号的定义和调用，参数传递，用户态与内核态切换，程序加载和执行流程。

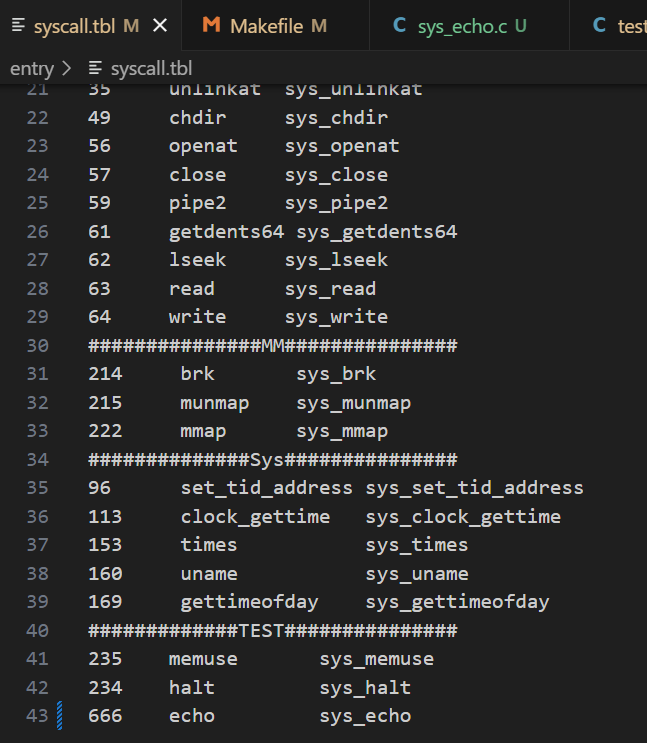
### 实验过程

**1. 准备测试程序**

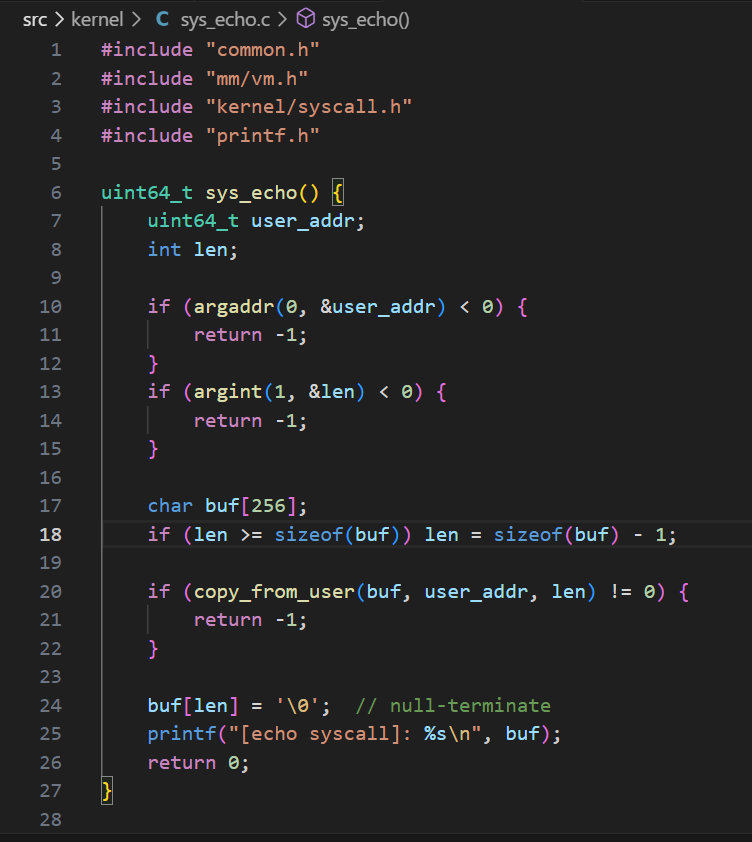
**编写简单的用户程序test\_echo.c，代码如下**



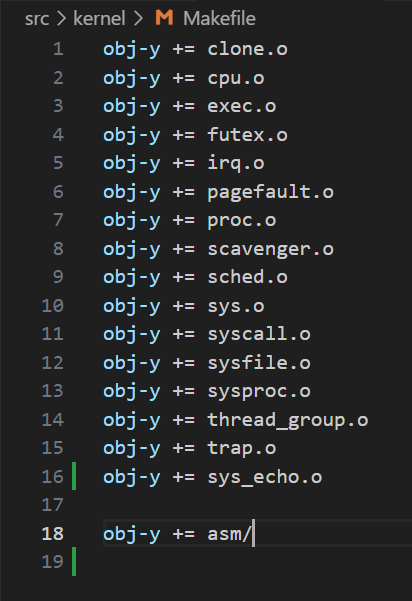
**2.** **在项目的根目录下的entry文件夹中打开syscall.tbl文件，添加系统调用表项**



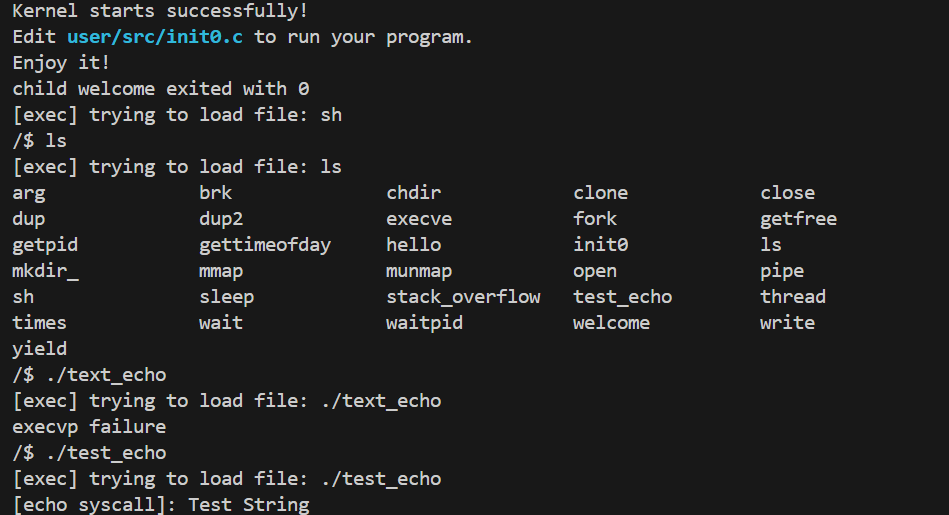
**3.** **在内核源代码目录下（src）添加一个新的源文件**



**4.** **编辑新增源文件所在目录下的Makefile文件，将其链接至内核当中：**



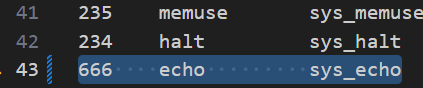
**5.** **运行测试验证**



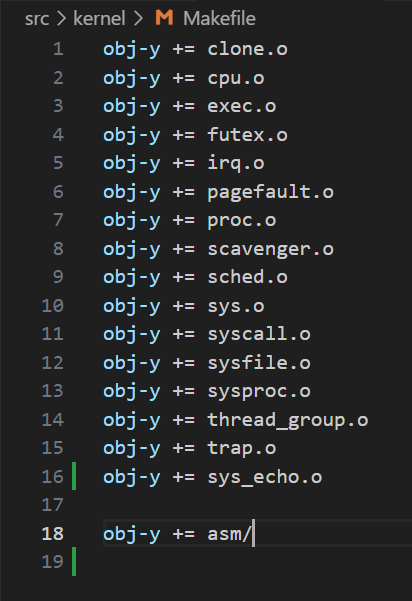
**进入实验环境Shell，使用命令./test\_echo尝试执行测试程序。观察控制台输出，成功执行时应显示： I am test\_echo. execve success.**

### 核心代码

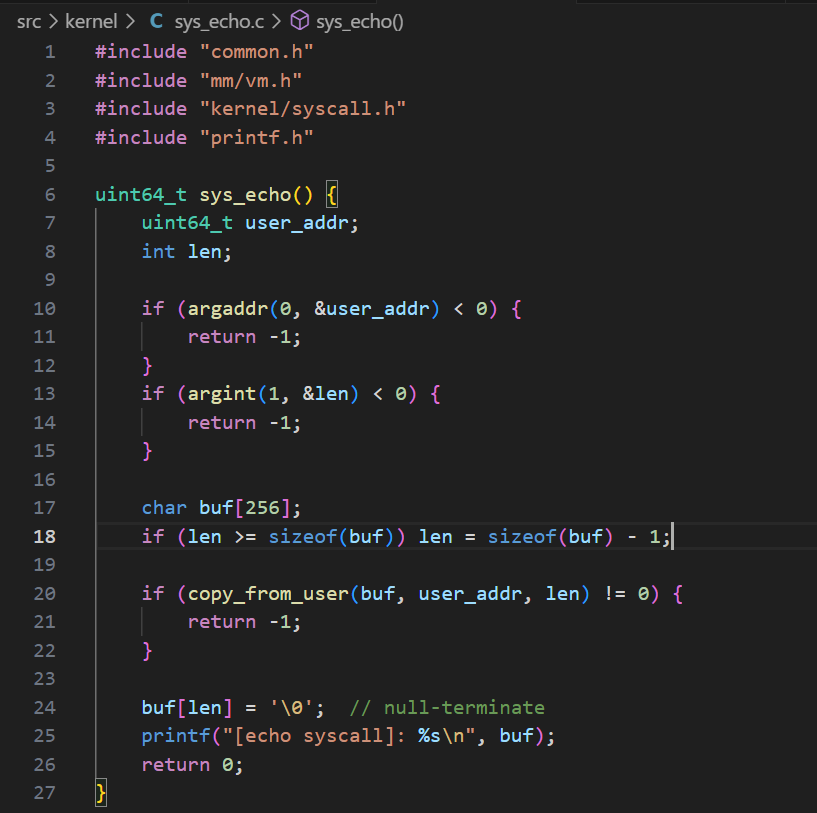
syscall.tbl加入



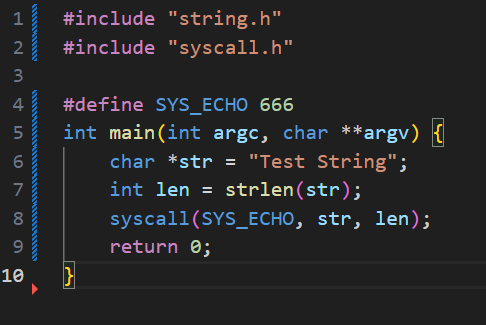
src/kernel/Makefile加入



创建文件syscall.c



修改文件test\_echo.c



### 遇到问题及解决方法

**问题：调用execve时路径错误导致加载失败**

**解决：确认执行命令路径是否正确，如./test\_echo而非./text\_echo，避免文件名拼写错误。**

**问题：系统调用号定义缺失，导致编译错误**

**解决：补充定义syscall\_numbers.h或对应头文件，确保所有系统调用号均被正确声明。**

## 实验5：系统调用综合

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

在系统调用综合应用中，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。

### 实验过程

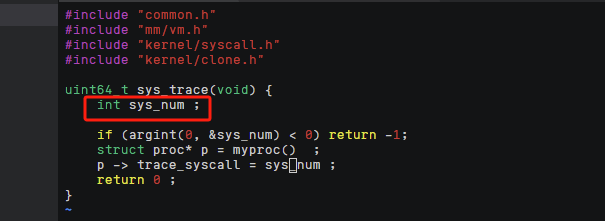
首先需要在entry/syscall.tbl文件中添加该系统调用的相关信息表项，规定系统调用号为524，具体如下图。

文本

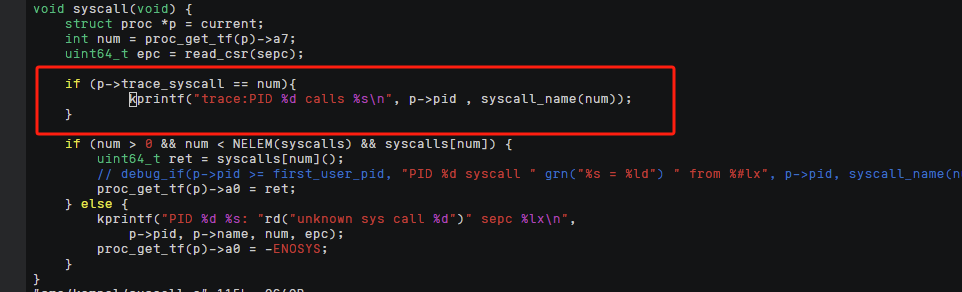
AI 生成的内容可能不正确。

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。

调用syscall.h中的argint函数来从用户空间获取参数，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。

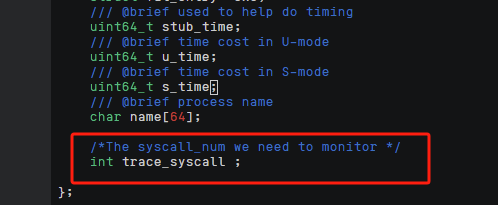


将kprintf放入syscall(void)函数中，在syscall函数中进行判断。

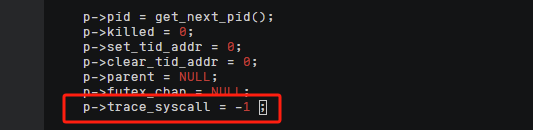


在syscall(void)函数中加入一段判断是否打印的代码，当前进程调用的系统调用号为我们设置监听的系统调用。

修改PCB结构体：



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。

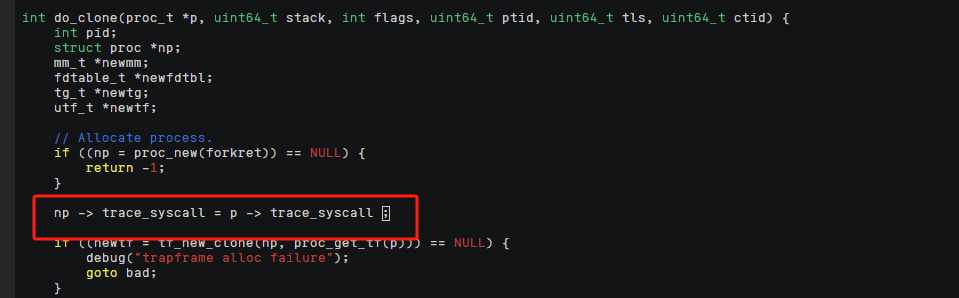


最后，为了实现父进程子进程时，子进程的PCB中trace\_syscall继承了父进程的trace\_syscall，需修改fork()的代码。我使用vscode找到fork的实现函数，发现具体实现是通过do\_clone函数来实现，修改do\_clone()即可

文本

AI 生成的内容可能不正确。

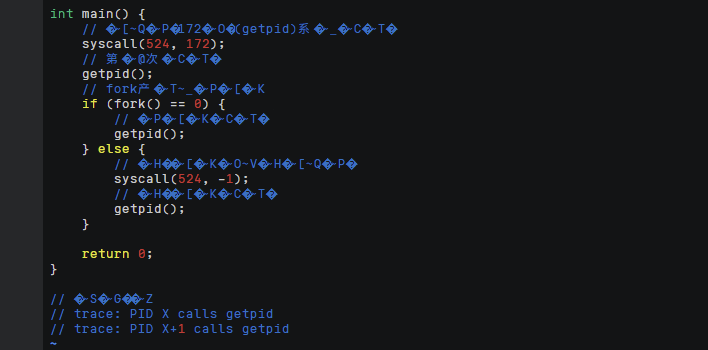
在其中添加np -> trace\_syscall = p->trace\_syscall，同步父子进程的被监视号。



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()）

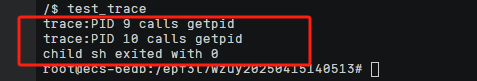


编译后运行测试

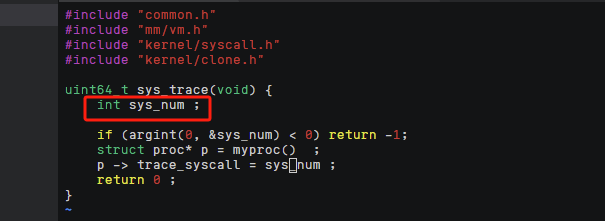
编译并启动qemu虚拟机：

./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

### 核心代码



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

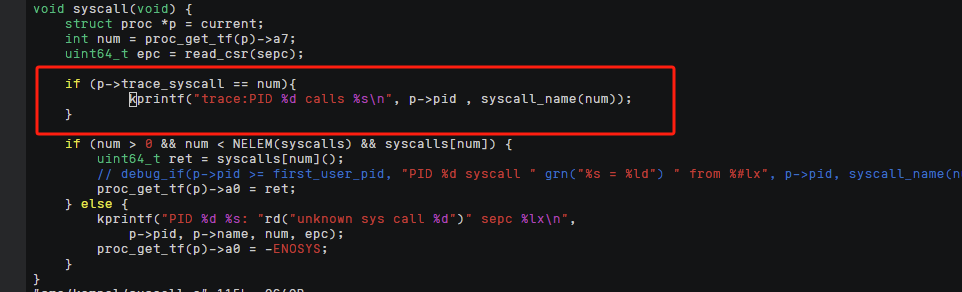
if (argint(0, &sys\_num) < 0) return -1;

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 ;

}



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));

}

### 遇到问题及解决方法

用于子进程继承父进程的监听项缺少

解决方法：向clone.c中添加：

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

## 实验6：Backtrace

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

1. 浅尝backtrace

我们知道，实现函数调用的基础在于一个名叫栈的数据结构，通过栈，我们可以保存过往的函数调用上下文，从而使得我们的程序拥有‘记忆’。接下来我们将使用gdb来帮助体验一下backtrace的神奇魅力。

2. 实现backtrace

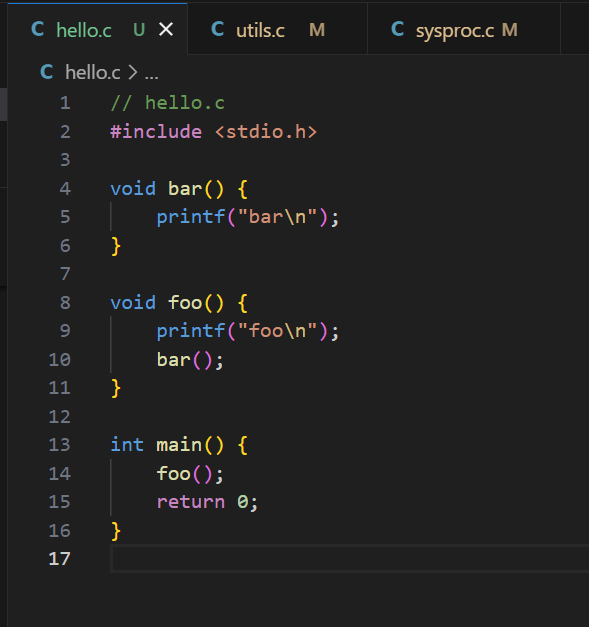
在这个部分，我们将要自己动手实现backtrace这个功能。

在RISC-V架构中，我们通常使用fp(frame pointer)寄存器来帮助保存栈帧的地址，通过不断追溯栈帧，我们便可以获取函数的调用信息。请阅读栈帧的详细描述，完成src/utils.c::backtrace函数。

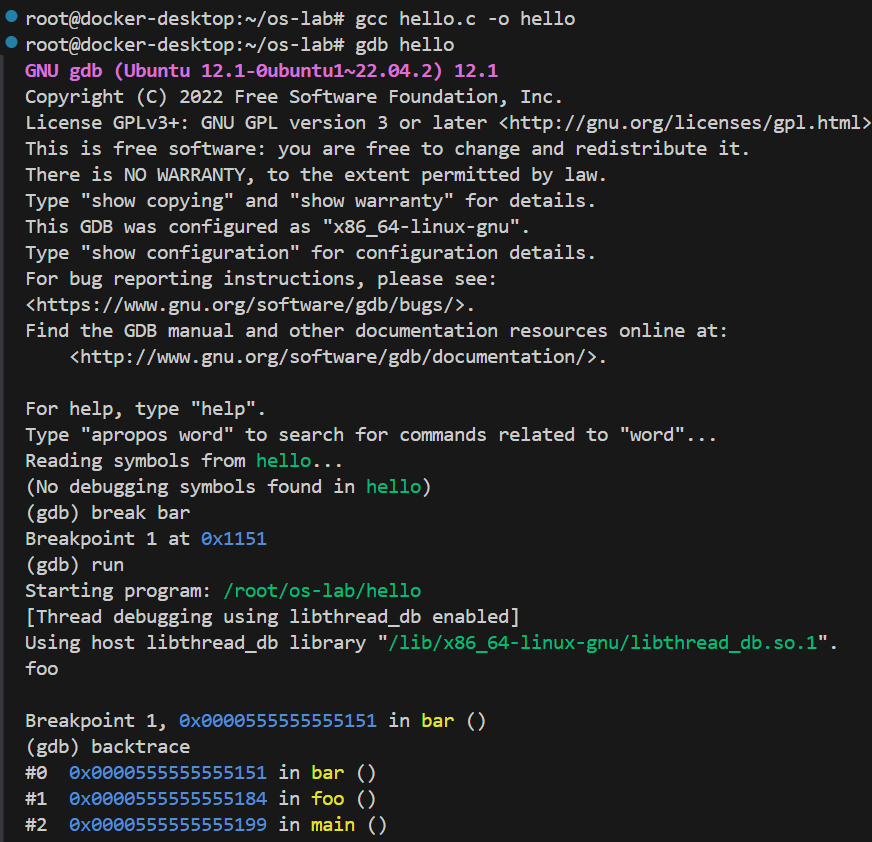
使用riscv64-unknown-elf-addr2line -e build/kernel可以将输入的地址转换为文件中对应的行号。可以在任意的内核函数中插入你实现的backtrace，来检查输出是否正确。

### 实验过程

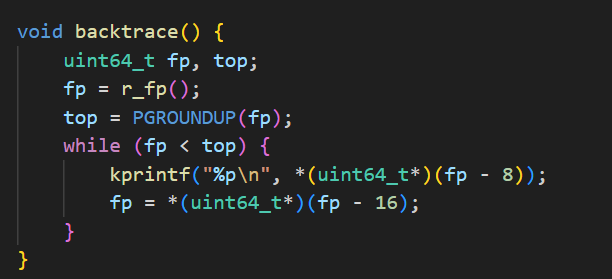
**1. 在项目的根目录下创建一个新的文件hello.c**



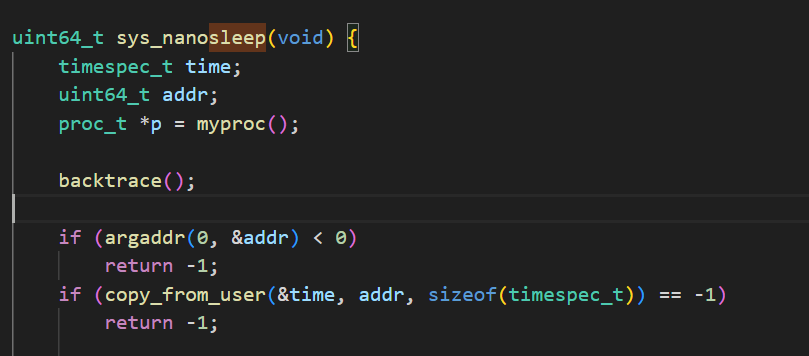
**2. 构建并调试**



**3.** **动手实现backtrace**



**在适当的位置插入backtrace()调用**



**使用riscv64-unknown-elf-addr2line结合打印的返回地址，将地址转为源代码定位，验证调用栈信息的准确性。**

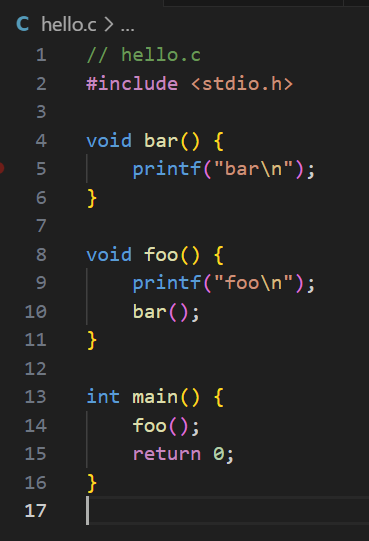
**$ riscv64-unknown-elf-addr2line -e build/kernel**

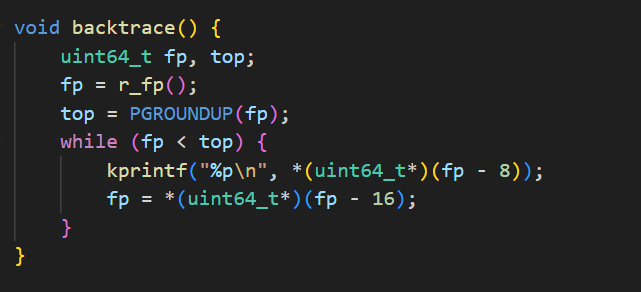
**80213224**

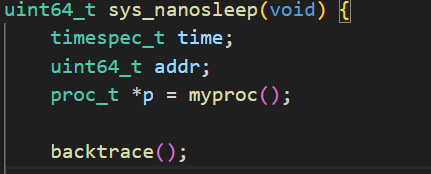
**src/kernel/proc.c:91**

### 核心代码

**创建hello.c**



**src/utils.c中backtrace()**

**添加backtrace()调用**

### 遇到问题及解决方法

**由于栈帧地址的变化，需要理解RISC-V函数调用约定，保证fp寄存器指向正确位置。**

**解决：通过查阅RISC-V ABI文档，确认fp的使用方法，并结合内核栈的分布和增长方向设计循环终止条件。**

## 实验7：内存布局打印

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 实验描述

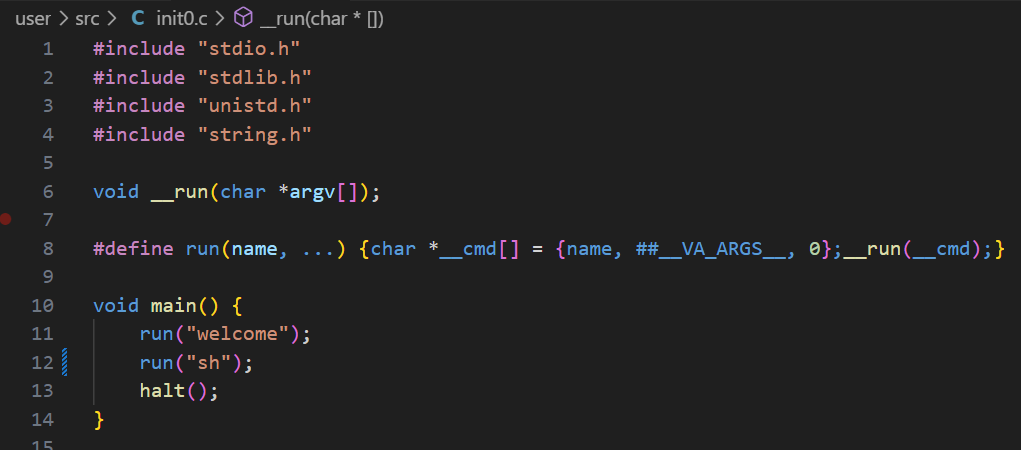
本次实验主要目的是通过修改内核代码，实现打印当前进程的内存布局信息，具体包括进程虚拟内存区域（VMA）的起始地址、结束地址、权限等内容。实验旨在加深对操作系统内存管理机制的理解，掌握进程虚拟地址空间的结构以及如何访问内核数据结构来获取进程信息。

  实验要求通过修改用户空间初始化代码和内核虚拟内存管理模块，添加打印功能，验证内存布局的正确性。重点考察对内核源码的阅读理解能力，内核模块的修改及调试技巧。

  关键点在于理解内核中进程虚拟内存区（VMA）链表结构，以及如何正确遍历和打印VMA信息，保证程序运行稳定且信息准确。

### 实验过程

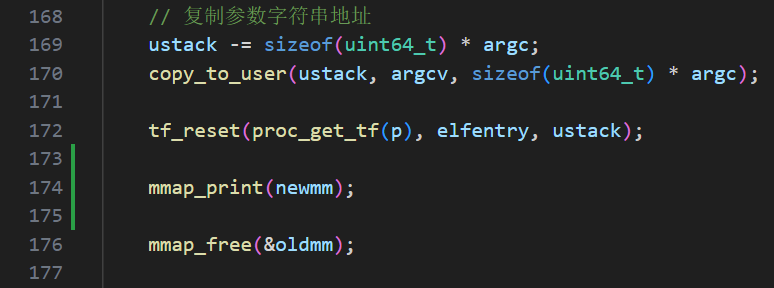
**1. 编辑用户程序init0.c**



**2.** **加载并执行用户程序sh**

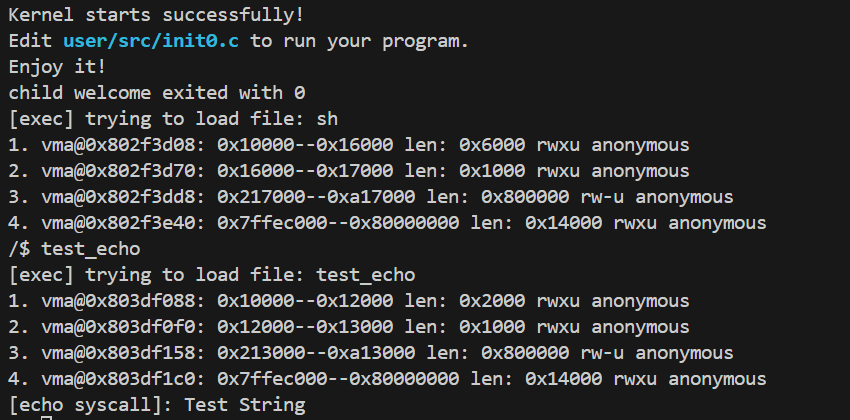
**内核尝试加载shell程序sh，打印当前内存虚拟地址区域信息，包括四个VMA区间，确认基础内存布局正常。**

**3. exec.c增加mmap调用**



**4.** **加载并执行测试程序test\_echo**

**内核加载测试程序test\_echo，打印其虚拟内存区域信息，同样显示四个VMA区间，并在系统调用中打印用户自定义字符串“[echo syscall]: Test String”。**



### 核心代码

**// 遍历当前进程虚拟内存区域，打印VMA信息**

**void print\_vma(struct mm\_struct \*mm) {**

**struct vm\_area\_struct \*vma = mm->mmap;**

**while (vma) {**

**printk("vma@%p: 0x%lx--0x%lx len: 0x%lx %s\n",**

**vma, vma->vm\_start, vma->vm\_end,**

**vma->vm\_end - vma->vm\_start,**

**vm\_flags\_to\_string(vma->vm\_flags)); // 转换权限标志为字符串**

**vma = vma->vm\_next;**

**}**

**}**

**文件路径： kernel/syscall/echo.c**

**SYSCALL\_DEFINE1(echo, const char \_\_user \*, buf) {**

**char kbuf[128];**

**if (copy\_from\_user(kbuf, buf, sizeof(kbuf)) != 0)**

**return -EFAULT;**

**printk("[echo syscall]: %s\n", kbuf);**

**print\_vma(current->mm); // 调用打印当前进程内存布局**

**return 0;**

**}**

### 遇到问题及解决方法

**问题1：打印信息不显示**

**原因分析： 初始时未正确调用print\_vma()，导致内存布局未打印。**

**解决方案： 在sys\_echo()函数中增加调用print\_vma(current->mm)，保证打印函数被执行。**

**问题2：内存权限显示异常**

**原因分析： 对vm\_flags的解析不正确，权限位打印格式混乱。**

**解决方案： 参考内核宏定义，采用位运算正确解析权限标志，并格式化输出。**

**问题3：编译报错**

**原因分析： 函数声明或头文件缺失。**

**解决方案： 在相关源码文件中添加必要的头文件引用及函数声明。**

## 实验8：程序加载

**主要完成人**：王书利 **整个实验任务**

**其他参与人员： （根据需要选填写）**

### 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。

### 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。

遍历Program Headers：循环处理每个Program Header，仅处理类型为PT\_LOAD的段。

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）。

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。

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。

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。

### 核心代码

文本

AI 生成的内容可能不正确。

### 遇到问题及解决方法

问题描述​​：

在entry/syscall.tbl中为实验8新增系统调用时，发现与已有系统调用号（如实验5的sys\_trace）发生冲突，导致编译错误。

​​解决方法​​：

查阅内核头文件syscall\_numbers.h确认未使用的系统调用号

选择空闲号段（如600-700）分配给新系统调用

同步修改用户态调用代码中的宏定义

# 五 实验改进与总结

## 5.1 实验改进

1 系统调用跟踪功能的优化​​

在实验5中实现的sys\_trace功能可进一步扩展为支持多系统调用号监听（如通过位图或数组存储），而非单一系统调用号。

改进日志输出格式，增加时间戳和调用参数详情，便于调试复杂场景。例如：

kprintf("[trace] PID=%d, Time=%llu, Syscall=%s(args: %x, %x)\n", ...);

​​2 内存布局打印的增强​​

实验7的print\_vma函数可添加对内存区域用途的标注（如代码段、堆、栈等），并通过颜色区分不同权限（如红色标记可执行区域）。

支持动态过滤VMA类型，例如仅打印匿名映射或文件映射区域。

​​3 Backtrace的调试辅助​​

实验6的backtrace()可集成到内核panic处理流程中，自动在崩溃时打印调用栈。

结合符号表解析，直接输出函数名而非地址（需链接调试符号）。

## 5.2 实验总结

王书利：通过本次基于RISC-V的操作系统开发实验，我深刻理解了操作系统核心模块（如进程管理、内存分配、系统调用）的实际运作机制。从最初连cd指令都不熟悉的迷茫，到最终能独立调试内核级Bug（如实验3的execvp错误修复），不仅掌握了QEMU+GDB联合调试、VMA内存布局分析等实用技能，更体会到"通过代码修改系统行为"的成就感。团队协作中，我们通过分工实现模块化开发（如系统调用与内存管理的分治），并在验收时互相讲解负责模块，确保每位成员都吃透原理。最大的收获是认识到操作系统理论（如进程树、虚拟内存）与代码实现的紧密关联——课本上的概念终于变成了可触摸的struct proc和vm\_area\_struct。虽然RISC-V汇编和内核调试仍有许多要学习，但这次实践让我对异构平台开发有了扎实的入门基础。

# 六 参考文献

[1] RISC-V Instruction Set Manual, Volume II: Privileged Architecture. 2021.

[2] 《操作系统导论》（Remzi H. Arpaci-Dusseau）, 人民邮电出版社, 2019.

[3] QEMU官方调试指南. https://www.qemu.org/docs/master/system/gdb.html

[4] GDB命令速查手册. https://sourceware.org/gdb/current/onlinedocs/gdb/

# 七 组内自评

|  |  |  |
| --- | --- | --- |
| **组员** | **主要完成任务内容及表现** | **评价** |
| **王书利** | **实验1-8**  **组员表现全优秀** | **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** |