



RISCV-Linux 内核分析实验手册

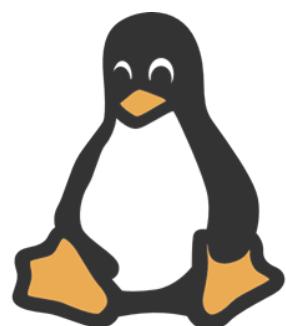
了解 RISCV LINUX 的入门之选

作者: Elon Li & ZyLqb

组织: Beijing Electronic Science and Technology Institute

时间: December 2, 2023

版本: 1.0



我们的征途是星辰大海——田中芳樹

*This manual is dedicated to Prof. Lou,
who has been tirelessly devoted to the field.*

前言

Linux 内核是一个开源、免费的操作系统内核，它是整个 Linux 操作系统的核心部分。但是由于 Linux 内核过于复杂，不易深入学习。由人民邮电出版社和中国工信出版社集团联合出版的《庖丁解牛 Linux 操作系统分析》一书，是我的恩师娄嘉鹏老师和孟宁老师共同编写的书籍并作为中科大软工学院和中办电科院研究生部的教材。自出版以来已获奖无数，该书内容翔实，通俗易懂，如今已在操作系统类图书排行上排名第六。

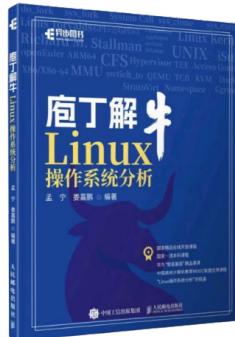


图 1：《庖丁解牛 Linux 操作系统分析》

本手册记录了该书中的八个实验，但与教材不同的是，本手册记录的八个实验均是在 RISC-V 架构上的机器上实现的。RISC-V（发音为“risk-five”）是一种基于精简指令集计算机（RISC）原则的开放架构，被视为将来最有潜力与 X86 架构和 Arm 架构相竞争的新指令集架构。在本手册中所记录的实验中，除了需要使用 GDB 调试的部分使用了 qemu 进行模拟 RISC-V 架构的裸机之外，均使用了来自赛芯科技研发的全球首发的新一代 RISC-V 架构的开发板——Visionfive 2 实现的。

本手册从内容上来说，记录了比较详细的过程，使得读者能够尽可能容易的复现出所有实验。通过阅读和复现本手册的各个实验，你除了将学习到《庖丁解牛 Linux 操作系统分析》中记录的关于 Linux 内核的知识点，还能学习到如何给 RISC-V 架构的开发板烧录 GNU/Linux 操作系统的发行版 Debain 操作系统；如何使用 ssh 和串口通讯连接 RISC-V 架构的开发板；OpenSBI 和 Uboot 是何物；一个简易的 Linux 操作系统将如何制作；如何使用 TFTP 协议给只带有 Uboot 的开发板传输文件；如何在 Uboot 上引导 Linux 内核和根文件系统；如何在 X86 平台上交叉编译 RISC-V Linux 内核；如何制作根文件系统；MenuOS 如何在 RISC-V 架构上实现；RISC Linux 内核是如何运行的等等。

本手册由我和 ZyLqb 共同完成，由于我们两个现在还都是学生，因此本手册中难免会出现各种错误，请各位读者批评指正。

目录

第1章 实验一：反汇编一个简单的 C 程序	1
1.1 反汇编一个简单的 C 程序	1
1.1.1 将 OS 烧录到 Micro-SD 卡上	1
1.1.2 ssh 连接 visionfive 2	1
1.2 反编译 c 语言代码	2
第2章 实验二：完成一个简单的时间片轮转多道程序内核代码	5
2.1 串口连接开发板	5
2.2 mincom 串口工具下载与设置	6
2.3 CH340 系列串口驱动占用	7
2.4 为编译 Linux 内核做准备	8
2.5 RISC-V 架构 MyKernel 内核的构建	8
2.5.1 实验目的和实验内容	8
2.5.2 了解 riscv	8
2.5.3 制作我们的简易调度器	10
2.6 编译 RISC-V 架构 MyKernel 内核	22
2.7 MyKernel 内核移植 VisionFive2 开发板	23
2.7.1 Ubuntu 安装和配置 TFTP 服务器	23
2.8 Uboot 加载 MyKernel 内核和根文件系统	23
2.8.1 配置 TFTP 服务	23
第3章 实验三：跟踪分析 Linux 内核的启动过程	25
3.1 下载 RISC-V 工具链	25
3.2 安装 QEMU	26
3.3 编译 OpenSBI	26
3.4 编译 Linux Kernel	27
3.5 制作根文件系统	28
3.6 运行简易 Linux 内核	29
第4章 实验四：使用库函数 API 和 C 代码中嵌入汇编代码两种方式使用同一个系统调用	36
4.1 使用 SSH 连接 starfive visionfive 2	36
4.2 使用 man 查看 write 函数	36
4.3 C 语言调用 write 函数	37
4.4 RISC-V 内联汇编调用 write 函数	37
4.5 内联汇编解释	38
第5章 实验五：分析 system call 中断处理过程	40
5.1 MenuOS 迁移到 RISC-V 架构	40
5.2 CWrite 和的编写	41
5.3 GDB 调试 sys_write 函数	43
第6章 实验六：分析 Linux 内核创建一个新进程的过程	48
6.1 阅读理解 task_struct 数据结构	48

6.2 分析 fork 函数对应的内核处理过程 sys_clone	48
6.3 GDB 跟踪分析 sys_clone	53
第 7 章 实验七：Linux 内核如何装载和启动一个可执行程	58
7.1 程序的编译过程	58
7.2 动态链接	62
7.3 gdb 跟踪分析一个 execve 系统调用	64
第 8 章 实验八：理解进程调度时机跟踪分析进程调度与进程切换的过程	69
8.1 理解 Linux 系统中进程调度的时机	69
8.2 gdb 跟踪分析 schedule() 函数	69
8.3 分析 switch_to 中的汇编代码	73
8.4 分析汇编代码	74
8.5 理解进程上下文的切换机制和中断上下文切换的关系	77

第1章 实验一：反汇编一个简单的 C 程序

1.1 反汇编一个简单的 C 程序

1.1.1 将 OS 烧录到 Micro-SD 卡上

现在我们需要将 Debian (Linux 发行版) 烧录到 Micro-SD 卡上，以便于它可以在 Visionfive 2 上运行。

使用 Micro-SD 卡读卡器或笔记本电脑上的内置读卡器，将 Micro-SD 卡连接至计算机。点击[链接](#)下载最新 Debian 镜像。解压.bz2 文件。访问[链接](#)下载 BalenaEtcher。我们将使用 BalenaEtcher 将 Debian 镜像烧录到 Micro-SD 卡上。

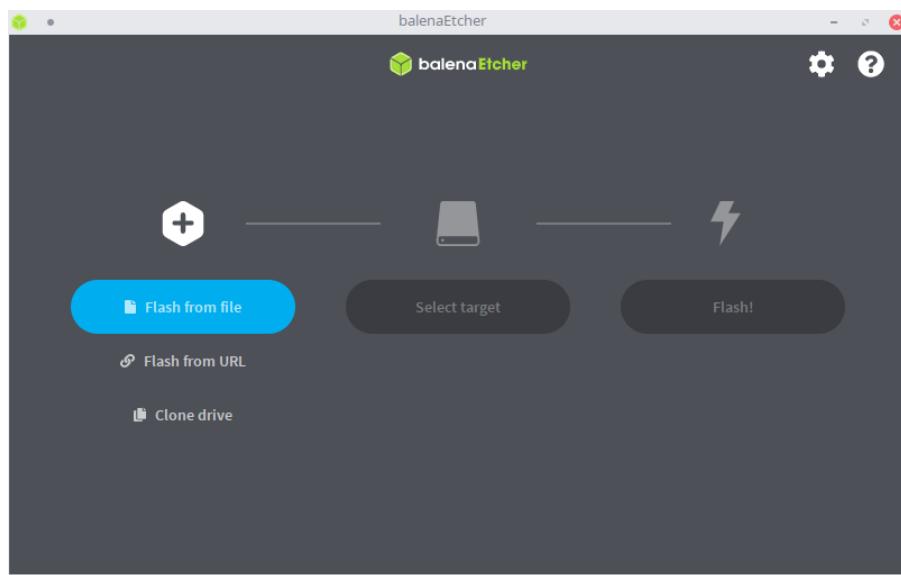


图 1.1: BalenaEtcher 界面

1.1.2 ssh 连接 visionfive 2

通过 HDMI 使用 Xfce 桌面环境登录，用户名和密码如下：

1. Username: root
2. Password: starfive

然后需要设置允许 root 用户通过 ssh 登录，通过使用以下命令进行修改：

```
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
```

修改完成后，使用以下命令进行重启 ssh 服务器：

```
/etc/init.d/ssh restart
```

使用浏览器登录到路由器地址，找到E · 星光 2的 IP 地址之后，通过 ssh 登录开发版，如下：

```
ssh root@192.168.1.xxx
```

完成此步后，需要输入密码才能完成登录。接下来到步骤需要使用 gcc 和 vim，通过使用以下命令进行安装 gcc 和 vim：

```
apt update  
apt install gcc vim
```

1.2 反编译 c 语言代码

安装好 gcc 和 vim 后，现在可以进行本次实验到核心操作——“反编译 c 语言代码”。首先使用 vim 创建 main.c 文件，使用下列的命令即可完成创建：

```
vim main.c
```

将下列示例代码复制以下代码到 main.c，需要注意的是在使用 vim 打开 main.c 后，vim 此时处于不可编辑到状态，需要轻点一次“i”键才能使用粘贴键。示例代码如下：

```
// main.c
int g(int x)
{
    return x + 3;
}

int f(int x)
{
    return g(x);
}

int main(void)
{
    return f(8) + 1;
}
```

完成好编辑任务后，使用以下的命令让 gcc 进行编译，使得 c 语言编译成为 RISCV 架构下汇编语言。命令如下：

```
gcc -S -o main.s main.c
```

得到 main.S 后，使用 vim 打开，内容如下：

```
.file "main.c"
.option pic
.text
.align 1
.globl g
.type g, @function
g:
    addi sp,sp,-32
    sd s0,24(sp)
    addi s0,sp,32
    mv a5,a0
    sw a5,-20(s0)
    lw a5,-20(s0)
    addiw a5,a5,3
    sext.w a5,a5
    mv a0,a5
    ld s0,24(sp)
    addi sp,sp,32
    jr ra
```

```

.size g, .-g
.align 1
.globl f
.type f, @function
f:
    addi sp,sp,-32
    sd ra,24(sp)
    sd s0,16(sp)
    addi s0,sp,32
    mv a5,a0
    sw a5,-20(s0)
    lw a5,-20(s0)
    mv a0,a5
    call g
    mv a5,a0
    mv a0,a5
    ld ra,24(sp)
    ld s0,16(sp)
    addi sp,sp,32
    jr ra
.size f, .-f
.align 1
.globl main
.type main, @function
main:
    addi sp,sp,-16
    sd ra,8(sp)
    sd s0,0(sp)
    addi s0,sp,16
    li a0,8
    call f
    mv a5,a0
    addiw a5,a5,1
    sext.w a5,a5
    mv a0,a5
    ld ra,8(sp)
    ld s0,0(sp)
    addi sp,sp,16
    jr ra
.size main, .-main
.ident "GCC: (Debian 11.3.0-3) 11.3.0"
.section .note.GNU-stack,"",@progbits

```

在 vim 的命令模式下，使用 :g/.*/d正则表达式删除带有的. 的行

```

g:
    addi sp,sp,-32
    sd s0,24(sp)
    addi s0,sp,32
    mv a5,a0

```

```
sw a5,-20($0)
lw a5,-20($0)
addiw a5,a5,3
mv a0,a5
ld $0,24($p)
addi $p,$p,32
jr ra
f:
addi $p,$p,-32
sd ra,24($p)
sd $0,16($p)
addi $0,$p,32
mv a5,a0
sw a5,-20($0)
lw a5,-20($0)
mv a0,a5
call g
mv a5,a0
mv a0,a5
ld ra,24($p)
ld $0,16($p)
addi $p,$p,32
jr ra
main:
addi $p,$p,-16
sd ra,8($p)
sd $0,0($p)
addi $0,$p,16
li a0,8
call f
mv a5,a0
addiw a5,a5,1
mv a0,a5
ld ra,8($p)
ld $0,0($p)
addi $p,$p,16
jr ra
```

实验一到此结束。

第2章 实验二：完成一个简单的时间片轮转多道程序内核代码

2.1 串口连接开发板

我们使用 USB 转 TTY 下载线连接 Starfive 旗下的 VisionFive2 开发板。主机操作系统是 Ubuntu22.04，具体配置如下。

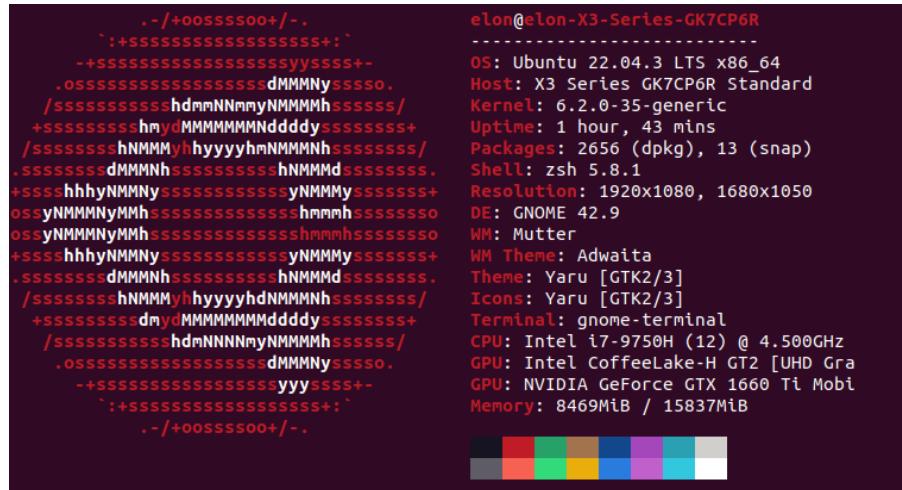


图 2.1: 系统配置

引脚连接实物图如下：

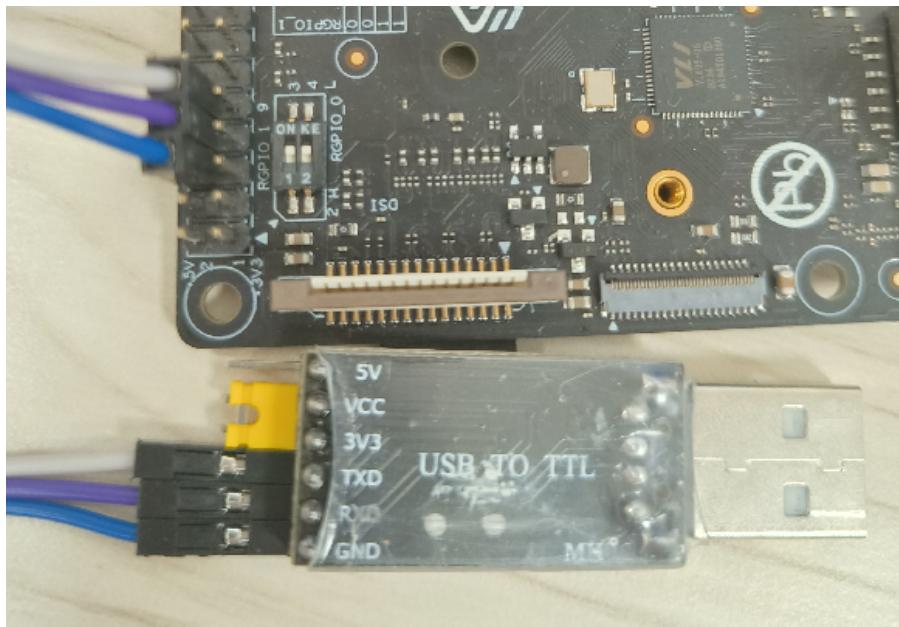


图 2.2: 引脚连接实物图

2.2 mincom 串口工具下载与设置

使用以下命令进行下载串口工具下载：

```
sudo apt update
sudo apt install minicom
```

使用以下命令对串口工具进行设置：

```
sudo minicom -s
```

进入 minicom 的设置界面后：方向键选择 Serial port setup 后按回车确定。

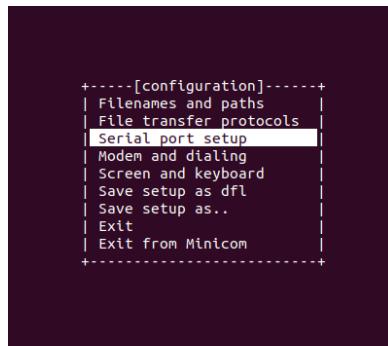


图 2.3: minicom 设置界面 1

进入串口设置界面后：按 Shift+a 选择 Serial Device, 将设备名改为 /dev/ttyUSB0.

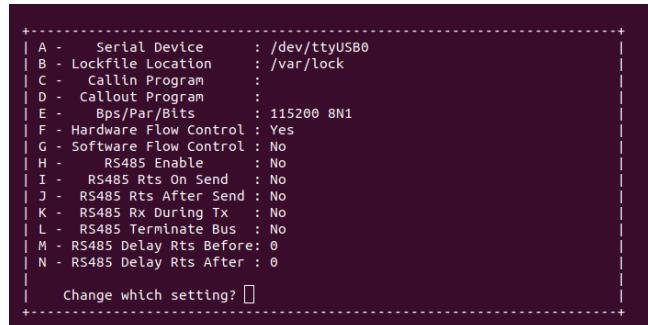


图 2.4: minicom 设置界面 2

修改完成后按下回车，返回上级菜单后，选择 Save setup as dfl 保存配置。

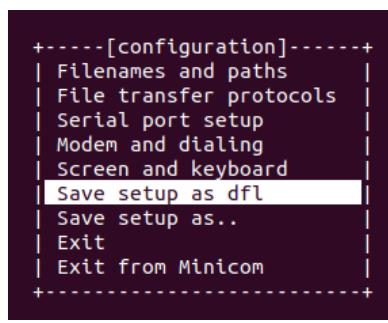


图 2.5: minicom 设置界面 3

保存后，选择 Exit 退出 Minicom

使用 Minicom 连接 VisionFive2 首先将连接好引脚的 USB 插口插入笔记本上，然后使用以下命令启动 Minicom。

```
sudo minicom
```

如果启动失败显示没有/dev/ttyUSB0，请参见下一节《CH340 系列串口驱动占用》。

启动后会显示进入 minicom。然后使用电源线连接开发板，给开发板加电。此时 Minicom 会开始显示加电后各种程序启动的信息：

```
Welcome to minicom 2.8
OPTIONS: I18n
Port /dev/ttyUSB0, 12:06:22
Press CTRL-A Z for help on special keys

U-Boot SPL 2021.10 (Sep 26 2023 - 22:53:52 +0800)
LPDDR4: 8G version: g8ad50857.
Trying to boot from SPI

OpenSBI v1.2
[REDACTED LOG]

Platform Name      : StarFive VisionFive V2
Platform Features  : medeleg
Platform HART Count : 5
Platform IPI Device   : aclkint-mswi
Platform Timer Device : aclkint-mtimer @ 4000000Hz
Platform Console Device : uart8250
Platform HSM Device   : ...
Platform PMU Device   : ...
Platform Reboot Device : pm-reset
Platform Shutdown Device : pm-reset
Platform Suspend Device : ...
Firmware Base       : 0x40000000
Firmware Size        : 392 KB
Firmware RW Offset   : 0x40000
Runtime SBI Version  : 1.0
```

图 2.6: minicom 启动

2.3 CH340 系列串口驱动占用

如果配置好 Minicom 后显示没有 ttyUSB 的问题，那么可能是由于 CH340 系列串口驱动被占用。

使用以下命令判断是否驱动被占用：

```
sudo dmesg | grep brltty
```

如果显示以下信息：

```
[ 7033.078452] usb 1-13: usbfs: interface 0 claimed by ch341 while 'brltty' sets config #1
```

说明：驱动占用

解决方法：使用以下命令卸载 brltty

```
sudo apt remove brltty
```

然后重新进行插拔 USB 接口，并使用以下的命令查看是否恢复正常：

```
ls /dev/ttyUSB0
```

若显示：

```
ls /dev/ttyUSB0
/dev/ttyUSB0
```

说明恢复正常，可以按照之前的步骤继续实验。

2.4 为编译 Linux 内核做准备

StarFive 为 VisionFive2 提供了一个 GitHub 仓库，阅读这个仓库中的 README 文件是本次实验的一部分。从 README 得知，为连编译能够在开发板上运行的 Linux 内核，首先需要为编译提供依赖环境。

使用以下的命令，在 ubuntu22.04 上创建可实现交叉编译 risc-v 架构的环境依赖：

```
sudo apt update
sudo apt-get install build-essential automake libtool texinfo bison flex gawk g++ git xxd curl wget
gdisk gperf cpio bc screen texinfo unzip libgmp-dev libmpfr-dev libmpc-dev libssl-dev libncurses
-dev libglib2.0-dev libpixman-1-dev libyaml-dev patchutils python3-pip zlib1g-dev device-tree-
compiler dosfstools mtools kpartx rsync
```

安装 Git LFS：

```
curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash
sudo apt-get install git-lfs
```

需要注意的是安装 Git LFS 需要访问外网，要完成此步骤需读者自行解决访问外网的问题或者使用国内镜像网站。

克隆 VisionFive2 仓库使用以下命令克隆 VisionFive2：

```
git clone https://github.com/starfive-tech/VisionFive2.git
cd VisionFive2
git checkout JH7110_VisionFive2-devel
git submodule update --init --recursive
```

切换分支：

```
cd buildroot && git checkout --track origin/JH7110_VisionFive2-devel && cd ..
cd u-boot && git checkout --track origin/JH7110_VisionFive2-devel && cd ..
cd linux && git checkout --track origin/JH7110_VisionFive2-devel && cd ..
cd opensbi && git checkout master && cd ..
cd soft_3rdpart && git checkout JH7110_VisionFive2-devel && cd ..
```

2.5 RISC-V 架构 MyKernel 内核的构建

2.5.1 实验目的和实验内容

这次实验主要目的是初识 RISC—V 架构。简单的了解一下 RISC-V 下的汇编，理解代码在 RISC-V 下是怎么跑起来的。我们将会实现一个非常简单的进程调度器，来帮助我们理解操作系统和 RISC-V。

2.5.2 了解 riscv

如果需要更加详细的了解请参考：

- RISC-V 中文参考
- The RISC-V Instruction Set Manual(Volume I , Volume II)
- ABI for RISC-V

我们这里只是简单的了解一下 riscv 寄存器的 RISC-V 应用程序二进制接口 (ABI)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

图 2.7: RISCV 寄存器

这些寄存器是 riscv 的通用寄存器，一共有 32 个，在这一章我们可能会用到 ra 寄存器：保存函数返回地址。

sp 寄存器：保存函数的栈指针。

函数是怎么执行的我们都知道，c 语言会被转化成汇编在转化成机器码。而汇编和机器码之间的转换是直接转换的。所以我们理解了一个函数在汇编上怎么运行的，那么我们就能理解函数在计算机上是如何运行的。

“我们这里使用的是 RISC-V 的指令架构，所以我们只讲解 RISC-V 下的汇编代码。但是为了更好的理解，我强烈建议先阅读理解 x86 下函数是怎么运行的。

- C 语言函数调用栈 (一)
- C 语言函数调用栈 (二)

”

首先我们来看在初入计算机世界的时候，我们遇到的第一份代码 hello world 的反汇编代码。

```
#include <stdio.h>
int main(){
    printf("hello world");
}
```

我们编译好之后用 objdump 反汇编一下，我这里去除掉了一些不需要的信息，只保留 main 函数的部分。

```
hello:      file format elf64-littleriscv

Disassembly of section .text:

000000000000062a <main>:
```

```

62a: 1141          addi   sp,sp,-16
62c: e406          sd     ra,8(sp)
62e: e022          sd     s0,0(sp)
630: 0800          addi   s0,sp,16
632: 00000517      auipc  a0,0x0
636: 07650513      addi   a0,a0,118 # 6a8 <_libc_csu_fini+0x6>
63a: f17ff0ef      jal    ra,550 <printf@plt>
63e: 4781          li     a5,0
640: 853e          mv    a0,a5
642: 60a2          ld     ra,8(sp)
644: 6402          ld     s0,0(sp)
646: 0141          addi   sp,sp,16
648: 8082          ret

```

我们来分析一下这段代码：

首先 addi sp,sp,-16 的作用是开栈（这里假定对函数栈有一定的了解，如果不了解请阅读上文的 c 语言函数调用栈），为函数开辟自己的栈帧。

然后是 sd ra,8(sp) 和 sd s0,0(sp) 这是将 ra 和 s0 分别存储在 sp+8 和 sp+0 的位置。这个里这个 ra 是函数的返回地址（调用者调用此函数的下一条指令）当函数执行完后，此函数的调用者时就是跳转到 ra 中存储的位置，如果此函数时最末端的叶子调用，是不用将其存入栈里面的。s0 是函数的栈的栈底，也就是帧指针。接下来是函数内部的一些计算跳转，我们先不管。

直接看到下面的 642 位置 ld ra,8(sp) 和 644ld s0.0(sp) 这两个指令是把函数在开始存储的返回地址和帧指针重新加载进相应的寄存器

然后是 addi sp,sp,-16 收回栈帧。最后 ret，这个 ret 是个伪指令，实际上是 jar ra

那么我们就可以大概了解函数是怎么运行的了：

1. 首先为函数开辟栈帧
2. 接着存储返回地址和上个栈帧的基址
3. 运算
4. 将存储的返回地址和帧地址重新加载进相应的寄存器
5. 收回栈帧
6. 返回上个函数调用此函数的位置的下一条指令。

2.5.3 制作我们的简易调度器

有了以上信息后，理论上我们已经可以手写汇编了，那么我们来完成一下我们的内容：编写一个简单的调度器。首先制定我们的需求：

- 可以进行进程切换
- 简单

那么一个进程里面会有什么呢—pc 寄存器加上通用寄存器，加上函数的栈帧。我们一般称之为函数现场。当现场没变，那么程序的状态就没变。也就是说当我们保存了现场，然后切换到其他进程运行一段时间后，恢复它那么我们就能切换回来，继续运行。那么我们就可以定义一个时间片段，允许每个进程运行一段时间然后切换到其他进程。这样我们就能做到根据时间片的多个进程的轮转调度了。

但是 riscv 的通用寄存器有 32 个，很多，我们其实是不需要全部保存的，只需要保存 caller save 的寄存器。但是对此时的我们来说还是有点多，我们可以再精简一点，省去我们不需要的寄存器，比如有关浮点数的寄存器我们这里是用不到的。

首先我们得有一个存储现场的地方，我们将现场存储在 pcb 的 AThread 里面，每次切换出去的时候把它保存进来，切换回来的时候，从这里把寄存器重新加载进去。

注意这里的 `_switch` 函数，在这里我们写了两部分的汇编代码。第一部分就是将当前的各种寄存器存入内存，第二部分就是将内存中存储的值加载进寄存器。但是函数入口会改变栈，所以在存储之前我们先要将函数入口给回退。

这部分的代码逻辑就很清晰了，定时器中断里面的值达到我们期望的值的时候进行一次进程切换。

```
//mypcb.h

/*
 *  linux/mykernel/mypcb.h
 *
 *  Kernel internal PCB types
 *
 *  Copyright (C) 2023 WangRui
 *
 */

#define MAX_TASK_NUM      4
#define KERNEL_STACK_SIZE 1024*2
/* CPU-specific state of this task */
//初始状态
typedef struct Thread {
    unsigned long    s0;
    unsigned long    ip;
    unsigned long    sp;
}tThread;
//保存的现场
typedef struct AThread {
    //unsigned long    s0;
    unsigned long    ra;
    unsigned long    sp;
    unsigned long    s0;
    unsigned long    s1;
    unsigned long    s2;
    unsigned long    s3;
    unsigned long    s4;
    unsigned long    s5;
    unsigned long    s6;
    unsigned long    s7;
    unsigned long    s8;
    unsigned long    s9;
    unsigned long    s10;
    unsigned long    s11;
}aThread;

typedef struct PCB{
    int pid;
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long stack[KERNEL_STACK_SIZE];//在这里存储进程的整个栈。
    /* CPU-specific state of this task */
```

```

    struct Thread thread;
    unsigned long task_entry;
    struct PCB *next;
}tPCB;

void my_schedule(void);

```

这部分代码主要是一些数据结构的定义，我们把一个进程的状态抽象成一个 pcb 块，在这里我们存入最主要的几个数据：

- pid：进程的进程号，是一个标识符，它是唯一的。
- state：用来表示进程当前的状态。如果是 runnable 的话就可以被进程调度器发现，并且参与调度。
- stack：程序运行时所使用的栈空间，每个程序的栈空间时不一样的。
- thread：用来保存程序运行时的寄存器状态，即现场。
- task_entry：用来表示程序的入口地址，简单理解相当于我们平时编程的 main 函数。
- next：表示下一个程序的地址。

```

// my_schedule
/*
 * linux/mykernel/myinterrupt.c
 *
 * Kernel internal my_timer_handler
 *
 * Copyright (C) 2023, 2023 WangRui
 *
 */
#include <linux/types.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/tty.h>
#include <linux/vmalloc.h>

#include "mypcb.h"

extern tPCB task[MAX_TASK_NUM];
extern tPCB * my_current_task;
extern volatile int my_need_sched;
volatile int time_count = 0;

/*
 * Called by timer interrupt.
 * it runs in the name of current running process,
 * so it use kernel stack of current running process
 */
//我们的定时器

void my_timer_handler(void)
{
    if(time_count%1000 == 0 && my_need_sched != 1)
    {

```

```

    printk(KERN_NOTICE ">>>my_timer_handler here<<<\n");
    my_need_sched = 1;
}
time_count ++ ;
return;
}

void __switch(aThread * old,aThread *new){
//old a0 new a1
asm volatile (
    "add s0,sp,-16\n"
    "ld s0,8(sp)\n"
    "add sp,sp,16\n"
    "sd ra,0(%[old]) \n"
    "sd sp,8(%[old]) \n"
    "sd s0,16(%[old]) \n"
    "sd s1,24(%[old]) \n"
    "sd s2,32(%[old]) \n"
    "sd s3,40(%[old]) \n"
    "sd s4,48(%[old]) \n"
    "sd s5,56(%[old]) \n"
    "sd s6,64(%[old]) \n"
    "sd s7,72(%[old]) \n"
    "sd s8,80(%[old]) \n"
    "sd s9,88(%[old]) \n"
    "sd s10,96(%[old]) \n"
    "sd s11,104(%[old]) \n"

    "ld ra, 0(%[new]) \n"
    "ld sp, 8(%[new]) \n"
    "ld s0, 16(%[new]) \n"
    "ld s1, 24(%[new]) \n"
    "ld s2, 32(%[new]) \n"
    "ld s3, 40(%[new]) \n"
    "ld s4, 48(%[new]) \n"
    "ld s5, 56(%[new]) \n"
    "ld s6, 64(%[new]) \n"
    "ld s7, 72(%[new]) \n"
    "ld s8, 80(%[new]) \n"
    "ld s9, 88(%[new]) \n"
    "ld s10, 96(%[new]) \n"
    "ld s11, 104(%[new]) \n"
    "ret \n"
    :
    : [old] "r" (old),[new] "r" (new)
);
}
//调度器

```

```

void my_schedule(void)
{
    tPCB * next;
    tPCB * prev;

    if(my_current_task == 0
       || my_current_task->next == 0)
    {
        return;
    }

    print("">>>>my_schedule<<<\n");
    /* schedule */
    next = my_current_task->next;
    prev = my_current_task;
    if(next->state == 0)/* -1 unrunnable, 0 runnable, >0 stopped */
    {
        my_current_task = next;
        print("stacks : %p\n",task->context.sp);
        print("">>>>switch %d to %d<<<\n",prev->pid,next->pid);
        __switch(&prev->context, &next->context);
        print("after scheduler\n");
    }
    return;
}

```

这里的 my_time_handler 函数时一个定时器中断所调用的函数，每当计算机产生一次定时器中断，就会调用这个函数。这个函数的作用是当触发了一定的定时器中断之后，就开启调度器。有了这个我们就能让我们的程序按时间片进行轮转运行了。而这里的 my_schedule 就是我们的调度器，他最主要的是 __switch 函数这个函数会将此时运行的程序的寄存器保存进 prev 结构体里面，然后将下一个进程的寄存器状态从 next 结构体里面取出来，并且加载进寄存器。

```

// 初始化
/*
 * linux/mykernel/mymain.c
 *
 * Kernel internal my_timer_handler
 *
 * Copyright (C) 2023, 2023 WangRui
 *
 */
#include <linux/types.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/tty.h>
#include <linux/vmalloc.h>

#include "mypcb.h"

```

```

tPCB task[MAX_TASK_NUM];
tPCB * my_current_task = NULL;
volatile int my_need_sched = 0;

void my_process(void);

void __init_my_start_kernel(void)
{

    print("run init my start kernel");
    int pid = 0;
    int i;
    /* Initialize process 0*/
    //unsigned long stacks = 0xffffffff81601000;

    task[pid].pid = pid;
    task[pid].state = 0; /* -1 unrunnable, 0 runnable, >0 stopped */
    task[pid].task_entry = task[pid].thread.ip = (unsigned long)my_process;
    task[pid].thread.sp = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZE-1];
    task[pid].thread.s0 = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZE-1];
    task[pid].next = &task[pid];

    task[pid].context.ra = (unsigned long)my_process;
    task[pid].context.s0 = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZE-1];
    task[pid].context.sp = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZE-1];

    /*fork more process */
    for(i=1;i<MAX_TASK_NUM;i++)
    {
        memcpy(&task[i],&task[0],sizeof(tPCB));
        task[i].pid = i;
        task[i].thread.sp = (unsigned long)(&task[i].stack[KERNEL_STACK_SIZE-1]);

        task[i].context.s0 = (unsigned long)&task[i].stack[KERNEL_STACK_SIZE-1];
        task[i].context.sp = (unsigned long)&task[i].stack[KERNEL_STACK_SIZE-1];

        task[i].next = task[i-1].next;
        task[i-1].next = &task[i];
    }
    print("stacks : %p\n",task[0].thread.sp);
    /* start process 0 by task[0] */
    pid = 0;
    my_current_task = &task[pid];
    __asm__ volatile(
        "mv sp,%[msp]\n"
        "mv ra,%[mip]\n"
        "mv s0,%[ms0]\n"
        "ret\n\t"           /* pop task[pid].thread.ip to rip */
    );
}

```

```

    :
    :[mip] "r" (task[pid].thread.ip),[msp] "r" (task[pid].thread.sp), [ms0]"r" (task[pid].thread.s0)
    /* input c or d mean %ecx/%edx*/
);
}

int i = 0;

void my_process(void)
{
    while(1)
    {
        i++;
        if(i%10000000 == 0)
        {
            printk(KERN_NOTICE "this is process %d -\n",my_current_task->pid);
            if(my_need_sched == 1)
            {
                my_need_sched = 0;
                my_schedule();
            }
            printk(KERN_NOTICE "this is process %d +\n",my_current_task->pid);
        }
    }
}
}

```

这里的 my_kernel_start 函数主要是进行一些初始化，其实主要就是初始化进程运行的栈空间，和 entry 的地址，这里的汇编代码的作用是将第一个程序的 entry 地址填入 ra 之中，而我们都知道 ra 是保存的是函数返回之后，下一条指令运行的地址，所以我们在这里将 entry 直接加载进 ra 然后跳转到这个地址，这样我们就能改变函数的运行顺序，从而进入我们自己写的程序里面。

Makefile:

```

obj-y      = mymain.o myinterrupt.o
kernel.patch:

diff --color -Naru Linux/drivers/clocksource/timer-riscv.c linux/drivers/clocksource/timer-riscv.c
--- Linux/drivers/clocksource/timer-riscv.c 2023-10-25 19:29:12.000000000 +0800
+++ linux/drivers/clocksource/timer-riscv.c 2023-10-29 16:10:34.294399984 +0800
@@ -20,6 +20,8 @@
#include <asm/smp.h>
#include <asm/sbi.h>
#include <asm/timex.h>
+//change
+#include "linux/timer.h"

static int riscv_clock_next_event(unsigned long delta,
        struct clock_event_device *ce)
@@ -86,7 +88,7 @@

```

```

csr_clear(CSR_IE, IE_TIE);
evdev->event_handler(evdev);

-
+ my_timer_handler();
  return IRQ_HANDLED;
}

diff --color -Naru Linux/include/linux/start_kernel.h linux/include/linux/start_kernel.h
--- Linux/include/linux/start_kernel.h 2023-10-25 19:29:16.000000000 +0800
+++ linux/include/linux/start_kernel.h 2023-10-29 16:16:36.018535237 +0800
@@ -7,7 +7,8 @@
/* Define the prototype for start_kernel here, rather than cluttering
   up something else. */

-
+//change
+extern void __init my_start_kernel(void);
extern asmlinkage void __init start_kernel(void);
extern void __init arch_call_rest_init(void);
extern void __ref rest_init(void);
diff --color -Naru Linux/include/linux/timer.h linux/include/linux/timer.h
--- Linux/include/linux/timer.h 2023-10-25 19:29:16.000000000 +0800
+++ linux/include/linux/timer.h 2023-10-29 16:19:31.419156143 +0800
@@ -191,7 +191,8 @@
#endif

#define del_singleshot_timer_sync(t) del_timer_sync(t)
-
+//change
+extern void my_timer_handler(void);
extern void init_timers(void);
struct hrtimer;
extern enum hrtimer_restart it_real_fn(struct hrtimer *);
diff --color -Naru Linux/init/main.c linux/init/main.c
--- Linux/init/main.c 2023-10-25 19:29:16.000000000 +0800
+++ linux/init/main.c 2023-10-29 16:22:14.563230772 +0800
@@ -1137,7 +1137,8 @@
    acpi_subsystem_init();
    arch_post_acpi_subsys_init();
    kcsan_init();
-
+ //change
+ my_start_kernel();
 /* Do the rest non-__init'ed, we're now alive */
 arch_call_rest_init();

diff --color -Naru Linux/Makefile linux/Makefile
--- Linux/Makefile 2023-10-25 19:29:11.000000000 +0800
+++ linux/Makefile 2023-10-29 16:24:49.013860022 +0800

```

```

@@ -1115,7 +1115,9 @@
export MODULES_NSDEPS := $(extmod_prefix)modules.nsdeps

ifeq ($(KBUILD_EXTMOD),)
-core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
+core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
+#change
+core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ mykernel/

vmlinux-dirs := $(patsubst %/,%,$(filter %/, \
$(core-y) $(core-m) $(drivers-y) $(drivers-m) \
diff --color -Nar u Linux/mykernel/Makefile linux/mykernel/Makefile
--- Linux/mykernel/Makefile 1970-01-01 08:00:00.000000000 +0800
+++ linux/mykernel/Makefile 2023-10-29 16:05:02.543702121 +0800
@@ -0,0 +1 @@
+obj-y = mymain.o myinterrupt.o
diff --color -Nar u Linux/mykernel/myinterrupt.c linux/mykernel/myinterrupt.c
--- Linux/mykernel/myinterrupt.c 1970-01-01 08:00:00.000000000 +0800
+++ linux/mykernel/myinterrupt.c 2023-10-29 17:58:31.532334640 +0800
@@ -0,0 +1,86 @@
+/*
+ * linux/mykernel/myinterrupt.c
+ *
+ * Kernel internal my_timer_handler
+ * Change IA32 to x86-64 arch, 2020/4/26
+ *
+ * Copyright (C) 2013, 2020 Mengning
+ *
+ */
+
+#include <linux/types.h>
+#include <linux/string.h>
+#include <linux/ctype.h>
+#include <linux/tty.h>
+#include <linux/vmalloc.h>
+
+#
+/#include "mypcb.h"
+
+extern tPCB task[MAX_TASK_NUM];
+extern tPCB * my_current_task;
+extern volatile int my_need_sched;
+volatile int time_count = 0;
+
+/*
+ * Called by timer interrupt.
+ * it runs in the name of current running process,
+ * so it use kernel stack of current running process
+ */
+
+void my_timer_handler(void)
+{

```

```

+     if(time_count%1000 == 0 && my_need_sched != 1)
+     {
+         printk(KERN_NOTICE ">>>my_timer_handler here<<<\n");
+         my_need_sched = 1;
+     }
+     time_count ++ ;
+     return;
+}

+
+void my_schedule(void)
+{
+    tPCB * next;
+    tPCB * prev;
+
+    if(my_current_task == NULL
+        || my_current_task->next == NULL)
+    {
+        return;
+    }
+    printk(KERN_NOTICE ">>>my_schedule<<<\n");
+    /* schedule */
+    next = my_current_task->next;
+    prev = my_current_task;
+    if(next->state == 0)/* -1 unrunnable, 0 runnable, >0 stopped */
+    {
+        my_current_task = next;
+        unsigned long prev_sp = prev->thread.sp;
+        unsigned long prev_ra = next->thread.ra;
+        unsigned long next_sp = next->thread.sp;
+        unsigned long next_ra = next->thread.ra;
+        printk(KERN_NOTICE ">>>switch %d to %d<<<\n",prev->pid,next->pid);
+        /* switch to next process */
+        // __asm__ volatile(
+        //     "mv %0,sp \n"
+        //     "mv %1,ra \n"
+        //     "ld sp,%2 \n"
+        //     "ld ra,%3 \n"
+        //     "ret"
+        //     : "=r" (prev->thread.sp), "=r" (prev->thread.ra)
+        //     : "m" (next->thread.sp), "m" ( next->thread.ra)
+        // );
+        __asm__ volatile (
+            "mv %0,sp \n"
+            "mv %1,ra \n"
+            :"=r" (prev_sp) , "=r" (prev_ra)
+        );
+        __asm__ volatile(
+
+            "mv sp,%0 \n"

```

```
+         "mv ra,%1 \n"
+
+         "ret"
+
+         :
+
+         : "r" (next_sp), "r" ( next_ra)
+
+     );
+
+ }
+
+ return;
+}

diff --color -Naru Linux/mykernel/mymain.c linux/mykernel/mymain.c
--- Linux/mykernel/mymain.c 1970-01-01 08:00:00.000000000 +0800
+++ linux/mykernel/mymain.c 2023-10-29 14:51:20.000000000 +0800
@@ -0,0 +1,75 @@
+/*
+ *  linux/mykernel/mymain.c
+ *
+ *  Kernel internal my_start_kernel
+ *  Change IA32 to x86-64 arch, 2020/4/26
+ *
+ *  Copyright (C) 2013, 2020 Mengning
+ *
+ */
+
+#include <linux/types.h>
+#include <linux/string.h>
+#include <linux/ctype.h>
+#include <linux/tty.h>
+#include <linux/vmalloc.h>
+
+
+
+/#include "mypcb.h"
+
+
+tPCB task[MAX_TASK_NUM];
+tPCB * my_current_task = NULL;
+volatile int my_need_sched = 0;
+
+
+void my_process(void);
+
+
+
+void __init my_start_kernel(void)
+{
+
+    int pid = 0;
+
+    int i;
+
+    /* Initialize process 0*/
+
+    task[pid].pid = pid;
+
+    task[pid].state = 0; /* -1 unrunnable, 0 runnable, >0 stopped */
+
+    task[pid].task_entry = task[pid].thread.ra = (unsigned long)my_process;
+
+    task[pid].thread.sp = (unsigned long)&task[pid].stack[KERNEL_STACK_SIZE-1];
+
+    task[pid].next = &task[pid];
+
+    /*fork more process */
+
+    for(i=1;i<MAX_TASK_NUM;i++)
+
```

```

+ {
+     memcpy(&task[i], &task[0], sizeof(tPCB));
+     task[i].pid = i;
+     task[i].thread.sp = (unsigned long)(&task[i].stack[KERNEL_STACK_SIZE-1]);
+     task[i].next = task[i-1].next;
+     task[i-1].next = &task[i];
+ }
+ /* start process 0 by task[0] */
+ pid = 0;
+ my_current_task = &task[pid];
+ __asm__ volatile(
+     "mv sp,%[sp]\n"
+     "mv ra,%[ra]\n"
+     "ret"
+ :
+ : [ra] "r" (task[pid].thread.ra), [sp] "r" (task[pid].thread.sp) /* input c or d mean %ecx/%edx*/
+ );
+}
+
+int i = 0;
+
+void my_process(void)
+{
+    while(1)
+    {
+        i++;
+        if(i%10000000 == 0)
+        {
+            printk(KERN_NOTICE "this is process %d -\n",my_current_task->pid);
+            if(my_need_sched == 1)
+            {
+                my_need_sched = 0;
+                my_schedule();
+            }
+            printk(KERN_NOTICE "this is process %d +\n",my_current_task->pid);
+        }
+    }
+}
+
diff --color -Naru Linux/mykernel/mypcb.h linux/mykernel/mypcb.h
--- Linux/mykernel/mypcb.h 1970-01-01 08:00:00.000000000 +0800
+++ linux/mykernel/mypcb.h 2023-10-29 14:51:27.000000000 +0800
@@ -0,0 +1,29 @@
+/*
+ * linux/mykernel/mypcb.h
+ *
+ * Kernel internal PCB types
+ *
+ * Copyright (C) 2013 Mengning
+ *

```

```

+ */
+
+#define MAX_TASK_NUM 4
+#define KERNEL_STACK_SIZE 1024*2
+/* CPU-specific state of this task */
+struct Thread {
+    unsigned long ra;
+    unsigned long sp;
+};
+
+typedef struct PCB{
+    int pid;
+    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
+    unsigned long stack[KERNEL_STACK_SIZE];
+    /* CPU-specific state of this task */
+    struct Thread thread;
+    unsigned long task_entry;
+    struct PCB *next;
+}}tPCB;
+
+void my_schedule(void);
+

```

使用以下命令进行补丁打包：

```
patch -p1 < ./kernel.patch
```

2.6 编译 RISC-V 架构 MyKernel 内核

将上一节生成的 linux 内核目录替换 VisionFive2 中的 linux 内核目录后，使用以下命令进行编译。

```
make -j$(nproc)
```

此次编译时间较长，编译成功后会在 VisionFive2 中自动创建 work 目录，所有的编译完成的文件都会在这个目录中生成。成功编译后，文件结构如图所示：

```

work/
  visionfive2-fw_payload.img
  image.fit
  initramfs.cpio.gz
  u-boot-spl.bin.normal.out
  linux/arch/riscv/boot
  dts
    starfive
      jh7110-visionfive-v2-ac108.dtb
      jh7110-visionfive-v2.dtb
      jh7110-visionfive-v2-wm8960.dtb
      vf2-overlay
        vf2-overlay uart3-i2c.dtbo
  Image.gz

```

2.7 MyKernel 内核移植 VisionFive2 开发板

2.7.1 Ubuntu 安装和配置 TFTP 服务器

使用以下命令安装 TFTP 服务器：

```
sudo apt-get install tftpd-hpa
```

查看 /etc/default/tftpd-hpa 文件：

```
sudo apt install vim
sudo vim /etc/default/tftpd-hpa
```

文件内容如下图所示：

```
# /etc/default/tftpd-hpa
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS=":69"
TFTP_OPTIONS="--secure"
```

图 2.8: tftpd-hpa 配置文件修改

明确 tftp 服务器的文件路径在”/srv/tftp”，将上一节生成的 work 目录下的所有文件拷贝到/srv/tftp 路径中。

2.8 Uboot 加载 MyKernel 内核和根文件系统

Uboot 加载 MyKernel 内核和根文件系统有两种方法加载。一种是以网络的形式，利用 tftp 协议和 uboot 命令直接将 MyKernel 内核和根文件系统加载到内存中，然后手动引导启动。另一种是通过烧录镜像到 TF 卡的方式。

首先讲解第一种方式：

按照《使用 Minicom 连接 VisionFive2》节中的步骤，连接 VisionFive2 和 Ubuntu 主机，待开发板成功引导 Uboot 后，使用下面的步骤进行配置。

2.8.1 配置 TFTP 服务

使用网线连接 VisionFive2 开发板后，通过路由器网管查看 Ubuntu 和 VisionFive2 开发板的 IP 地址。并使用以下的命令设置 TFTP 服务器的

```
setenv serverip 192.168.1.101;
setenv ipaddr 192.168.1.28;
setenv getewayip 192.168.1.1;
```

在 Uboot 中下载 image.fit 文件，下载速度取决于局域网的带宽，请耐心等待：

```
tftpboot ${loadaddr} image.fit;
```

使用以下命令手动引导内核和根文件系统：

```
bootm start ${loadaddr};
bootm loados ${loadaddr};
run chipa_set_linux;
run cpu_vol_set;
booti ${kernel_addr_r} ${ramdisk_addr_r}:${filesize} ${fdt_addr_r};
```

第二种方式相较于第一种方式，几乎不需要在开发板上有任何命令输入。

首先需要准备一张 TF 卡和读卡器，在 VisionFive2 目录中的使用以下命令生成 SD 卡 Image 文件：

```
sudo make -j$(nproc)
sudo make buildroot_rootfs -j$(nproc)
sudo make img
```

运行完成之后，将在 work 目录下产生 sdcard.img 文件。

使用 BalenaEtcher 将 work 目录中的生成的 sdcard.img 文件烧录至 TF 卡中后，即可在开发板实现 MyKernel 内核和根文件系统的加载。

MyKernel 内核加载后将在开发板上实现一个简单的时间片轮转多道程序，如图下所示：

```
this is process 0 -
>>>my_schedule<<
>>>switch 0 to 1<<
this is process 1 -
>>>my_schedule<<
>>>switch 1 to 2<<
this is process 2 -
>>>my_schedule<<
>>>switch 2 to 3<<
this is process 3 -
>>>my_schedule<<
>>>switch 3 to 0<<
after scheduler
this is process 0 +
this is process 0 -
>>>my_schedule<<
>>>switch 0 to 1<<
after scheduler
this is process 1 +
this is process 1 -
>>>my_schedule<<
>>>switch 1 to 2<<
after scheduler
this is process 2 +
this is process 2 -
>>>my_schedule<<
>>>switch 2 to 3<<
after scheduler
this is process 3 +
this is process 3 -
>>>my_schedule<<
>>>switch 3 to 0<<
after scheduler
this is process 0 +
```

图 2.9：一个简单的时间片轮转多道程序

第3章 实验三：跟踪分析 Linux 内核的启动过程

3.1 下载 RISC-V 工具链

对于新手而言，自己克隆 RISC-V 的仓库进行编译有三大不方便之处：

1. RISC-V 工具链仓库巨大，对于国内用户下载不方便
2. RISC-V 工具链的依赖包和配置，新手不一定能解决
3. RISC-V 工具链编译时间长

故此，使用他人编译好的工具链，是一种有效的方式。网站 toolchains.bootlin.com 提供了已经编译好的 RISC-V 工具链，如下图所示：

The screenshot shows the 'About' page of the toolchains.bootlin.com website. On the left, there is a sidebar with links to 'About', 'Toolchains', 'News', and 'FAQ'. The main content area has a heading 'Download' and two dropdown menus: 'Select arch' set to 'riscv64-lp64d' and 'Select libc' set to 'glibc'. Below these are two large download buttons: 'Download stable' and 'Download bleeding-edge'. Each button has a circular arrow icon above it. To the right of each button is a small green circle with a checkmark and the text 'Tests passed' followed by 'checksum (sha256)'. Below the download buttons is a table comparing stable and bleeding-edge versions for various components. At the bottom of the page, there is a link 'View all riscv64-lp64d toolchains'.

Component	stable	bleeding-edge
binutils	2.40	2.41
gcc	12.3.0	13.2.0
gdb	12.1	13.2
glibc	2.37-2-g9f8513d...	2.37-2-g9f8513d...
linux-headers	5.4.251	5.10.188

图 3.1: RISC-V 工具链

选项：

1. 在 Select arch 选项中，我们选择 riscv64-lp64d
2. 在 Select libc 中选择 glibc
3. 下载 stable 版或者 Bleeding-edge

本节以下载 Bleeding-edge 为例。首先使用以下的命令在 目录下创建一个名为的目录：

```
mkdir riscv64_oslab  
cd riscv64_oslab
```

下载之后使用以下的命令进行解压：

```
wget https://toolchains.bootlin.com/downloads/releases/toolchains/riscv64-lp64d/tarballs/riscv64-lp  
64d--glibc--bleeding-edge-2023.08-1.tar.bz2  
tar -jxvf riscv64-lp64d--glibc--bleeding-edge-2023.08-1.tar.bz2
```

解压之后，使用你喜欢的编辑器打开位于 目录下的.bashrc 或者.zshrc 设置工具链的环境变量，下面将使用 emacs 在 zsh 下设置工具链的环境变量：

```
emacs ~/.zshrc
```

在.zshrc 中添加以下语句：

```
export PATH=/home/elon/riscv64_oslab/riscv64-1p64d--glibc--bleeding-edge-2023.08-1/bin:$PATH
```

需要注意的是 /home/elon 需要更换为自己的用户名，在 riscv64_oslab/riscv64-1p64d--glibc--bleeding-edge-2023.08-1/bin 目录中可使用 pwd 命令显示自己需要添加的路径。

3.2 安装 QEMU

qemu 是一个开源且免费的硬件虚拟化仿真器，可以提供不同的虚拟的计算机架构。我们使用 qemu 在 x86 平台上模拟 rsicv 架构的裸机用于调试和测试 linux 内核。

在riscv64_oslab目录下使用以下的命令可以下载和解压 qemu:

```
wget https://download.qemu.org/qemu-7.1.0.tar.xz  
tar xvJf qemu-7.1.0.tar.xz
```

解压后使用以下命令进行配置和编译以及安装：

```
cd qemu-7.1.0  
./configure  
make -j$(nproc)  
make install
```

其中make -j\$(nproc)中的-j\$(nproc)参数为以机器硬件线程数进行多线程编译。

3.3 编译 OpenSBI

SBI 是 RISC-V 架构下的特权层二进制接口，是用于引导程序环境的规范。通俗的讲就是 x86 下的 bios，但这并不准确，如果想要详细了解什么是 OpenSBI，可以参考这个链接 [OpenSBI Deep Dive](#)。OpenSBI 的英文全称是 RISC-V Open Source Supervisor Binary Interface。我们使用 OpenSBI 引导 linux 内核的加载。

使用以下的命令在riscv64_qlab目录下克隆 OpenSBI:

```
git clone https://github.com/riscv-software-src/opensbi.git
```

使用以下命令进行编译：

```
export CROSS_COMPILE=riscv64-linux-  
make PLATFORM=generic -j$(nproc)
```

最后生成的 OpenSBI 固件在 build/platform/generic/firmware/ 目录下产生：

```
riscv64_ostab cd opensbi  
→ opensbi git:(master) ls build/platform/generic/firmware  
fw_dynamic.bin fw_dynamic.o fw_jump.elf.ld fw_payload.elf.dep  
fw_dynamic.dep fw_jump.bin fw_jump.o fw_payload.elf.ld  
fw_dynamic.elf fw_jump.dep fw_payload.bin fw_payload.o  
fw_dynamic.elf.dep fw_jump.elf fw_payload.dep payloads  
fw_dynamic.elf.ld fw_jump.elf.dep fw_payload.elf  
→ opensbi git:(master) 
```

图 3.2: 产生的 OpenSBI 固件

生成的目录下有三个关键词需要解释：

1. dynamic: 带有动态信息的固件
 2. jump: 指定下一级的 boot 地址跳转
 3. payload: 包含下一级 boot 的二进制内容, 通常是 uboot/linux

为了减少工作量，我们不编译 uboot，所以我们选择 jump 关键字的固件 - `fw_jump.elf`.

下图演示了，我们启动 Linux 内核的次序，与常规方式不同的在于我们使 OpenSBI 直接 jumps 跳到 Linux Kernel 的启动处。

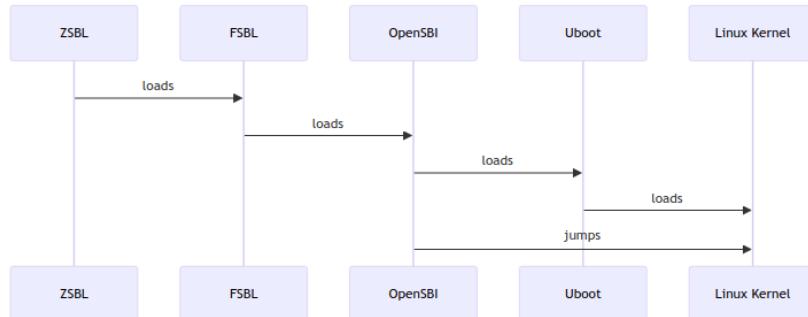


图 3.3: OpenSBI 引导操作系统

3.4 编译 Linux Kernel

由于 Linux Kernel 在 6.0 的版本后使用了 rust 语言进行了编写，为了减少不必要的麻烦和意外，我们使用的 Linux 内核版本小于 v6.0。

使用以下命令在 `riscv64_oslalb` 目录下下载和解压 Kernel:

```

wget http://mirrors.besti.net/kernel/v5.x/linux-5.19.16.tar.xz
tar -xf linux-5.19.16.tar.xz
cd linux-5.19.16

```

需要注意的是我们使用了电科院的内部 Linux Kernel 的镜像网站进行下载，读者需要自己的寻找国内的镜像。

解压之后，需要指定编译 Linux Kernel 的架构和方式。为了编译出 RISC-V 架构的 Linux 内核，我们需要指定编译为 RISC-V 的架构，同时由于我们是在 X86 的平台上进行编译，所以我们必须使用交叉编译的方式进行编译。命令如下：

```

export ARCH=riscv
export CROSS_COMPILE=riscv64-linux-
make defconfig

```

使用 `make defconfig` 后 Linux 内核目录下会产生一个 `.config` 文件，为了方便后面的使用 GDB 调试 RISC-V 版本的 Linux 内核，我们需要为我们编译的内核附加上调试信息。我们需要修改 Linux 内核目录下的 `Makefile` 文件，这里依旧使用 `emacs` 进行修改，读者请使用自己喜好的编辑器修改。

```
emacs Makefile
```

在 `emacs` 中使用 `Ctrl+s` 粘贴 `KBUILD_CFLAGS` 按下回车，找到 `KBUILD_CFLAGS` 的位置，在后面的选项中加入 `-g`。例如图下：

此举是为了提供 GDB 调试的功能。修改完成后保存即可退出，进行最后的编译：

```
make -j$(nproc)
```

```

521 KBUILD_AFLAGS := -D__ASSEMBLY__ -fno-PIE
522 KBUILD_CFLAGS := -Wall -g -Wundef -Werror=strict-prototypes -Wno-trigraphs \
523   -fno-strict-aliasing -fno-common -fshort-wchar -fno-PIE \
524   -Werror=implicit-function-declaration -Werror=implicit-int \
525   -Werror=return-type -Wno-format-security \
526   -std=gnu11
527 KBUILD_CPPFLAGS := -D__KERNEL__
528 KBUILD_AFLAGS_KERNEL :=
529 KBUILD_CFLAGS_KERNEL :=

```

图 3.4: 修改 Makefile

编译完成后将在两个地方生成文件。一处就在在 linux 内核目录下, 另一处在 linux 内核目录下的 arch/riscv /boot/, 如图所示:

第一处:

```

$ ls
arch      fs       lib           modules.order  System.map
block     include  LICENSES      Module.symvers  tools
certs     init     MAINTAINERS  net             usr
COPYING   io_uring Makefile     README          virt
CREDITS   ipc     mm           samples         vmlinux
crypto    Kbuild  modules.builtin scripts        vmlinux.o
Documentation Kconfig modules.builtin.modinfo security
drivers   kernel  modules-only.symvers sound        vmlinux.symvers

```

图 3.5: linux 内核目录

第二处:

```

$ ls arch/riscv/boot
dts  Image  Image.gz  install.sh  loader.lds.S  loader.S  Makefile

```

图 3.6: boot 内核目录

3.5 制作根文件系统

一般一个简易的 Linux 操作系统包括两个部分, 一个是 Linux Kernel, 另一个是根文件系统。制作根文件系统通常有两个工具可以使用, 一种是 BusyBox, 另一种是 Buildroot。使用 BusyBox 制作根文件系统的步骤较多, 但是很灵活。而 Buildroot 操作简单。

首先在 riscv64_oslab 目录下下载和解压 buildroot:

```

wget https://buildroot.org/downloads/buildroot-2023.02.6.tar.gz
tar -xvf buildroot-2023.02.6.tar.gz

```

解压后进入 buildroot 目录进行 buildroot 配置:

```

cd buildroot-2023.02.6
make menuconfig

```

使用上述命令将出现一个菜单界面。首先, 进入 Target options。

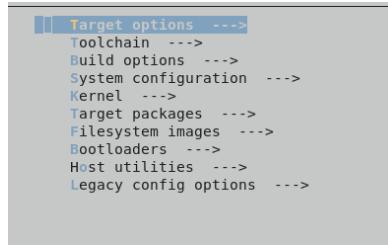


图 3.7: Target options

选择 Target Architecture 为 RISCV。

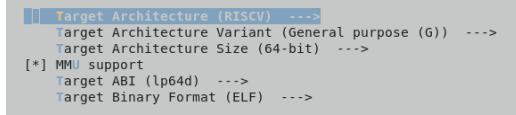


图 3.8: 选择 RISCV

Exit 返回一级界面后，选择 Filesystem images 后选择 ext2/3/4 root filesystem。

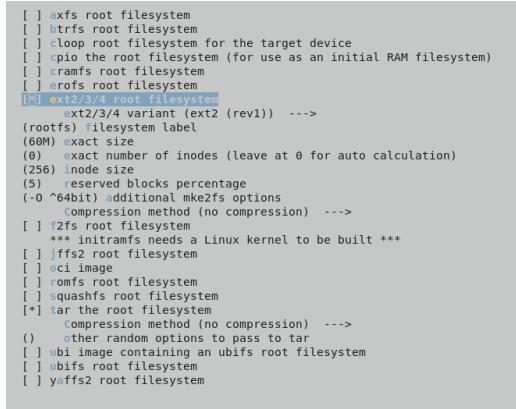


图 3.9: 选择 root filesystem

保存后使用以下命令进行编译：

```
make -j$(nproc)
```

本次编译由于 buildroot 需要编译自己的编译需要的工具链，因此时间较长，请读者耐心等待。编译完成后，在 output/images 目录下将生成我们需要的文件。

至此，漫长的编译过程已经结束了。下一节，我们将运行自己制作的一个简易 linux 操作系统和进行 GDB 远程调试内核，从 start_kernel 到 init 进程启动。

3.6 运行简易 Linux 内核

使用以下命令在 riscv64_oslalb 目录创建 images 目录：

```
mkdir images
```

使用以下命令在 images 目录下将之前各个部分的编译好的文件复制到 images 目录下：

```
cp ./opensbi/build/platform/generic/firmware/fw_jump.elf .
cp ./linux-5.19.16/vmlinux .
cp ./linux-5.19.16/arch/riscv/boot/Image .
cp ./buildroot-2023.02.6/output/images/rootfs.ext2 .
```

```
→ buildroot-2023.02.6 ls output/images
rootfs.ext2  rootfs.tar
→ buildroot-2023.02.6 └─
```

图 3.10: output files

复制完后 images 目录下将产生以下文件:

```
→ images ls
fw_jump.elf  Image  rootfs.ext2  rootfs.ext2.bak  start-gdb.sh  start-qemu.sh  vmlinuz
→ images └─
```

图 3.11: images output files

现在需要编译一个名叫 start-qemu.sh 的 shell 脚本。

使用你喜欢的编辑器打开 start-qemu.sh，填写下以下内容。这里仍然使用 emacs 作为示例：

```
emacs start-qemu.sh
```

内容如下：

```
#!/bin/sh

qemu-system-riscv64 -M virt \
-bios fw_jump.elf \
-kernel Image \
-append "rootwait root=/dev/vda ro" \
-drive file=rootfs.ext2,format=raw,id=hd0 \
-device virtio-blk-device,drive=hd0 \
-netdev user,id=net0 -device virtio-net-device,netdev=net0 -nographic
```

使用以下命令给 start-qemu.sh 赋予执行权：

```
chmod +x start-qemu.sh
```

使用以下命令即可运行简易的 RISC-V 架构的简易 Linux 操作系统：

```
./start-qemu.sh
```

启动界面如下：



图 3.12: 启动界面

加载完后显示用户登录：

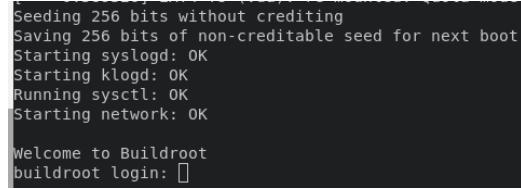


图 3.13: 用户登录

输入 root 点击回车，进入 shell。

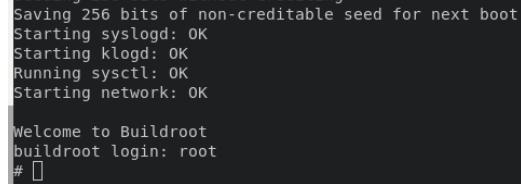


图 3.14: 进入 shell

从 start_kernel 到 init 进程启动上节，我们已经完成了在 qemu 上运行一个简易的 RISC-V 架构的 Linux 操作系统。这一节，将对这个 Linux 内核进行调试。

使用你喜欢的编辑器创建 start-gdb.sh shell 脚本，按照惯例，笔者依旧使用 emacs。

```
emacs ./start-gdb.sh
```

脚本内容如下：

```
qemu-system-riscv64 -M virt \
-bios fw_jump.elf \
-kernel Image \
-append "rootwait root=/dev/vda ro" \
-drive file=rootfs.ext2,format=raw,id=hd0 \
-device virtio-blk-device,drive=hd0 \
-netdev user,id=net0 -device virtio-net-device,netdev=net0 \
-nographic \
-s -S
```

保存后，使用以下的命令对 start-gdb.sh 赋予执行权：

```
chmod +x ./start-gdb.sh
```

使用以下命令安装 gdb-multiarch 为接下来的调试做准备：

```
sudo apt install gdb-multiarch
```

现在我们开始正式调试 RISC-V 架构的 Linux 内核。

首先在 images 目录下使用以下命令，启动 qemu 并开启远程调试：

```
./start-gdb.sh
```

然后打开另一个终端，进入 images 目录后使用以下命令进入 gdb：

```
gdb-multiarch ./vmlinux
```

进入 gdb 后显示：Reading symbols from ./vmlinux... 说明操作正常，可以进一步操作：

```
→ images gdb-multiarch ./vmlinux
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vmlinux...
(gdb) □
```

图 3.15: 进入 gdb

现在在 gdb 内输入以下命令后回车：

```
target remote:1234
```

此举是为了建立 gdb 和在 qemu 中启动的 gdbserver 之间的连接。连接完毕后显示如下：

```
Reading symbols from ./vmlinux...
(gdb) target remote:1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) □
```

图 3.16: 建立 gdb 连接

说明建立连接成功。

现在使用以下命令 gdb 对 start_kernel 进行打断点:

```
break start_kernel
```

gdb 显示 start_kernel 函数在 init/main.c 文件中。

```
0x0000000000001000 in ?? ()
(gdb) break start_kernel
Breakpoint 1 at 0xffffffff808006b4: file init/main.c, line 930.
(gdb) 
```

图 3.17: gdb 显示 start_kernel 函数

使用 c 命令让程序执行到 start_kernel 断点处:

```
(gdb) c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:930
warning: Source file is more recent than executable.
930 {
(gdb) 
```

图 3.18: 执行到 start_kernel 断点处

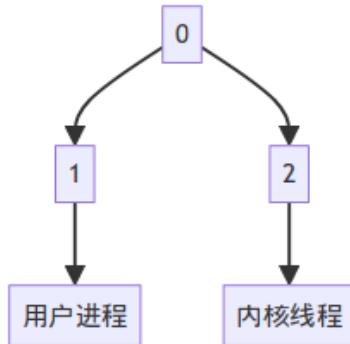
现在使用 l 命令显示 start_kernel 断点处:

```
Breakpoint 1, start_kernel () at init/main.c:930
warning: Source file is more recent than executable.
930 {
(gdb) l
925             &unknown_options[1]);
926         memblock_free(unknown_options, len);
927     }
928
929     asmlinkage __visible void __init __no_sanitize_address start_kernel(void)
930     {
931         char *command_line;
932         char *after_dashes;
933
934         set_task_stack_end_magic(&init_task);
(gdb) 
```

图 3.19: 显示 start_kernel 断点处信息

l 命令显示 start_kernel 函数位于 init/main.c 中的 930 行。

在 Linux 内核的进程树中，涉及到三个重要的进程，分别是 0 号进程和 1 号进程以及 2 号进程。示意图如下所示：



0 号进程是通过手动创建的，在 `start_kernel` 中出现的 `&init_task` 就是 0 号进程的进程描述符。

接下来，让我们先使用编辑器打开 `linux-5.19.16` 中 `init` 目录中的 `main.c` 文件找到 `asmlinkage __visible void __init __no_sanitize_address start_kernel(void)`。让我们一起阅读 `start_kernel` 函数的 Linux 内核源码。

在这部分的源码中的 1138 行，也就是 `start_kernel` 函数的倒数第二个函数，有一个名叫 `arch_call_rest_init()` 的函数。`arch_call_rest_init()` 的定义如下：

```

void __init __weak arch_call_rest_init(void)
{
    rest_init();
}
  
```

我们在 `gdb` 中不断使用 `n` 命令，当 `next` 到到了 `arch_call_rest_init` 时，内核即将创建 `kernel_init` 1 号进程和 `kthreadd` 2 号进程。

```

1130      poking_init();
(gdb) n
1134      arch_post_acpi_subsys_init();
(gdb) n
1138      arch_call_rest_init();
(gdb) 
  
```

图 3.21: `arch_call_rest_init`

使用b rest_init 对rest_init()函数进行打断点，进入rest_init 函数后如下：

```
-init/main.c
 680
 681     noinline void __ref rest_init(void)
 682     {
 683         struct task_struct *tsk;
 684         int pid;
 685
B+> 686         rCU_SCHEDULER_STARTING();
 687         /*
 688          * We need to spawn init first so that it obtains pid 1, however
 689          * the init task will end up wanting to create kthreads, which, if
 690          * we schedule it before we create kthreadd, will OOPS.
 691          */
 692         pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
 693
 694         * Pin init on the boot CPU. Task migration is not properly working
 695         * until sched_init_smp() has been run. It will set the allowed
B+> 0xffffffff806a5ae2 <rest_init>      addi    sp,sp,-16
 0xffffffff806a5ae4 <rest_init+2>      sd      ra,8(sp)
 0xffffffff806a5ae6 <rest_init+4>      sd      s0,0(sp)
 0xffffffff806a5ae8 <rest_init+6>      addi    s0,sp,16
 0xffffffff806a5aea <rest_init+8>      auipc   ra,0xffff9c8
 0xffffffff806a5aee <rest_init+12>     jalr    -1586(ra)
 0xffffffff806a5af2 <rest_init+16>     li      a2,512
 0xffffffff806a5af6 <rest_init+20>     li      a1,0
 0xffffffff806a5af8 <rest_init+22>     auipc   a0,0x0
 0xffffffff806a5afc <rest_init+26>     addi    a0,a0,206
 0xffffffff806a5b00 <rest_init+30>     auipc   ra,0xffff968
 0xffffffff806a5b04 <rest_init+34>     jalr    -1586(ra)
 0xffffffff806a5b08 <rest_init+38>     lw      a5,8(tp) # 0x8
 0xffffffff806a5b0c <rest_init+42>     addiw   a5,a5,1
 0xffffffff806a5b0e <rest_init+44>     sw      a5,8(tp) # 0x8
 0xffffffff806a5b12 <rest_init+48>     auipc   a1,0xb6e
```

图 3.22: rest_init

此时执行下一步会出现调用 `user_mode_thread`，此时这个函数就是创建 1 号进程，而 1 号进程也被称为 init 进程。继续向下单步执行，会发现调用 `kernel_thread`，这个函数就是用于创建 2 号进程，即内核线程。

此时再使用 c 命令，将完成 Linux Kernel 引导，系统进入登录界面：

图 3.23: 进入登录界面

至此调试结束。

`start-kernel` 函数中包括了内核的初始化和一系列的设置，包括文件系统、调度器、定时器、内存管理、锁初始化、内存加密、NUMA 策略、ACPI 初始化等等。而 `rest_init` 函数宣告了初始化的结束。

第4章 实验四：使用库函数 API 和 C 代码中嵌入汇编代码两种方式使用同一个系统调用

4.1 使用 SSH 连接 starfive visionfive 2

在第一个实验中，我们配置好了使用 ssh 连接 starfive visionfive 2 开发板的流程。在连接好网线后，通过使用浏览器登录网管地址查看开发板的 ip 地址后，使用以下命令连接 starfive visionfive 2 开发板。

```
ssh root@192.168.xxx.xxx
```

注意：192.168.xxx.xxx Ip 地址需要使用自己查阅的 Ip 地址。

回车后，输入密码：starfive。进入 starfive visionfive 2 开发板中的 shell。如下图所示：

```
→ ~ ssh root@192.168.1.102
root@192.168.1.102's password:
Linux starfive 5.15.0-starfive #1 SMP Wed Aug 23 11:18:20 CST 2023 riscv64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Nov  5 04:31:35 2023 from 192.168.1.100
root@starfive:~#
```

图 4.1: starfive visionfive 2 Shell

4.2 使用 man 查看 write 函数

使用以下命令查看 write 函数的具体用法

man 2 write 具体用法如图所示：

```
WRITE(2)                               System Calls Manual                               WRITE(2)

NAME
    write - write to a file descriptor

LIBRARY
    Standard C library (libc, -lc)

SYNOPSIS
    #include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
    write() writes up to count bytes from the buffer starting at buf to the file
    referred to by the file descriptor fd.

    The number of bytes written may be less than count if, for example, there is
    insufficient space on the underlying physical medium, or the RLIMIT_FSIZE re-
    source limit is encountered (see setrlimit(2)), or the call was interrupted by
    a signal handler after having written less than count bytes. (See also
    pipe(7).)

    For a seekable file (i.e., one to which lseek(2) may be applied, for example,
    a regular file) writing takes place at the file offset, and the file offset is
    incremented by the number of bytes actually written. If the file was
    open(2)ed with O_APPEND, the file offset is first set to the end of the file
    before writing. The adjustment of the file offset and the write operation are
    performed as an atomic step.

    POSIX requires that a read(2) that can be proved to occur after a write() has
    returned will return the new data. Note that not all filesystems are POSIX
```

图 4.2: man 2 write

write 函数有三个参数。第 0 个参数，输出位置；第 1 个参数，指向输出内容的指针；第 2 个参数，输出内容的长度。

4.3 C 语言调用 write 函数

接下来，我们将在 shell 中 vim 编写 local_write.c 程序，通过使用 C 语言调用 write 函数。

首先我们需要下载 vim，使用以下命令下载安装 vim。

```
sudo apt install vim
```

安装好后，使用以下命令创建 local_write.c 文件。

```
vim local_write.c
```

将下面的代码复制到 local_write.c 中：

```
#include <stdio.h>
#include <unistd.h>
int main(void){
    char s[]="hello, world.\n";
    write(1,s,13);
    return 0;
}
```

使用以下命令进行编译和运行：

```
gcc -o local_write -c local_write.c
./local_write
```

显示以下信息说明运行成功：



```
root@starfive:~# ./local_write
hello, world.root@starfive:~#
```

图 4.3: local_wirte 运行成功

4.4 RISC-V 内联汇编调用 write 函数

接下来，我们将使用 RISC-V 内联汇编嵌入 C 语言代码中，调用 write 函数。

使用以下命令创建 asm_write.c 文件。

```
vim asm_write.c
```

将下面的代码复制到 asm_write.c 中：

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char s[] = "hello, world\n";
    __asm__ volatile(
```

```

"li a2, 13\n"
"li a0, 1\n"
"mv a1, %[str]\n"
"li a7, 64\n"
"ecall \n"
:
: [str] "r" (s)
);

return 0;
}

```

使用以下命令进行编译和运行：

```

gcc -o asm_write -c asm_write.c
./asm_write

```

显示以下信息说明运行成功：

图 4.4: asm_write 运行成功

4.5 内联汇编解释

可能最开始有很多不了解的人会认为系统调用和函数调用差不多，然而当我们用汇编代码运行了第一个系统调用之后就会发现这个步骤和函数调用还是很不同的，即使他们在你是使用库函数的时候感觉是差不多的。它没有函数的入口，也没有函数的出口（请看 lab2 中的资料），而且还多了一个 ecall。那么系统调用是怎么运行的呢？

抽象系统资源为了完全明白这段汇编代码，让我们先了解一点前置的知识。

当谈及操作系统时，人们可能会问的第一个问题是为什么需要它？也就是说，我们可以将所有的系统调用实现为一个库，应用程序可以与之链接。在此方案中，每个应用程序甚至可以根据自己的需求定制自己的库。应用程序可以直接与硬件资源交互，并以应用程序的最佳方式使用这些资源（例如，实现高性能或可预测的性能）。一些嵌入式设备或实时系统的操作系统就是这样组织的。

这种库函数方法的缺点是，如果有多个应用程序在运行，这些应用程序必须表现良好。例如，每个应用程序必须定期放弃中央处理器，以便其他应用程序能够运行。如果所有应用程序都相互信任并且没有错误，这种协同操作的分时方案可能是可以的。然而更典型的情况是，应用程序互不信任且存在 bug，所以人们通常希望提供比合作方案更强的隔离。

为了实现强隔离，最好禁止应用程序直接访问敏感的硬件资源，而是将资源抽象为服务。例如，Unix 应用程序只通过文件系统的 open、read、write 和 close 系统调用与存储交互，而不是直接读写磁盘。这为应用程序提供了方便实用的路径名，并允许操作系统（作为接口的实现者）管理磁盘。即使隔离不是一个问题，有意交互（或者只是希望互不干扰）的程序可能会发现文件系统比直接使用磁盘更方便。

同样，Unix 在进程之间透明地切换硬件处理器，根据需要保存和恢复寄存器状态，这样应用程序就不必意识到分时共享的存在。这种透明性允许操作系统共享处理器，即使有些应用程序处于无限循环中。

另一个例子是，Unix 进程使用 exec 来构建它们的内存映像，而不是直接与物理内存交互。这允许操作系统决定将一个进程放在内存中的哪里；如果内存很紧张，操作系统甚至可以将一个进程的一些数据存储在磁盘上。exec 还为用户提供了存储可执行程序映像的文件系统的便利。

linux 系统调用接口是精心设计的，既为程序员提供了便利，又提供了强隔离的可能性。Unix 接口不是抽象资源的唯一方法，但它已经被证明是一个非常好的方法

用户态，内核态，以及系统调用强隔离需要应用程序和操作系统之间的硬边界，如果应用程序出错，我们不希望操作系统失败或其他应用程序失败，相反，操作系统应该能够清理失败的应用程序，并继续运行其他应用程序，要实现强隔离，操作系统必须保证应用程序不能修改（甚至读取）操作系统的数据结构和指令，以及应用程序不能访问其他进程的内存。

CPU 为强隔离提供硬件支持。例如，RISC-V 有三种 CPU 可以执行指令的模式：机器模式 (Machine Mode)、用户模式 (User Mode) 和管理模式 (Supervisor Mode)。在机器模式下执行的指令具有完全特权；CPU 在机器模式下启动，OpenSBI 也是运行在机器模式下的。操作系统运行在管理模式，用户程序运行在用户模式。

在管理模式下，CPU 被允许执行特权指令：例如，启用和禁用中断、读取和写入保存页表地址的寄存器等。如果用户模式下的应用程序试图执行特权指令，那么 CPU 不会执行该指令，而是切换到管理模式，以便管理模式代码可以终止应用程序，因为它做了它不应该做的事情。应用程序只能执行用户模式的指令（例如，数字相加等），并被称为在用户空间中运行，而此时处于管理模式下的软件可以执行特权指令，并被称为在内核空间中运行。在内核空间（或管理模式）中运行的软件被称为内核。

想要调用内核函数的应用程序（例如上面的 write 系统调用）必须过渡到内核。CPU 提供一个特殊的指令，将 CPU 从用户模式切换到管理模式，并在内核指定的入口点进入内核（RISC-V 为此提供 ecall 指令）。一旦 CPU 切换到管理模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝它或执行它。由内核控制转换到管理模式的入口点是很重要的；如果应用程序可以决定内核入口点，那么恶意应用程序可以在跳过参数验证的地方进入内核。

write 调用的汇编代码现在我们已经明白了使用系统调用的原因。那么理解起来这段汇编代码就很简单了。首先我们需要将参数填入合适的位置，write 系统调用的第一个参数是写在哪，需要我们传入文件描述符。第二个参数是写入的字符串的指针。第三个参数是写入字符串的长度。我们将每个值按照相应的方法

```
"li a2, 13\n"
"li a0, 1\n"
"mv a1, %[str]\n"
```

加载进寄存器（这部分是和函数调用是一样的），然后接下来我们需要陷入到内核态，在 RISC-V 里面我们使用的是 ecall。我们将 ecall 写上，但是此时系统并不知道我们是什么系统调用，所以我们需要在 a7 寄存器里面加入我们的系统调用号 64。自此系统调用的代码我们就完成了。

第5章 实验五：分析 system call 中断处理过程

5.1 MenuOS 迁移到 RISC-V 架构

MenuOS 是接下来四次实验的平台。但是由于 MenuOS 原本设计是为 x86 架构所服务的，如果需要在 RISC-V 架构上的机器上运行，就必须对关键部分的汇编代码进行修改。

首先使用以下命令克隆 MenuOS：

```
cd ~/riscv64_oslab/
mkdir MenuOS
cd MenuOS
git clone https://github.com/mengning/menu.git
```

克隆后，进入 menu 目录，使用你喜欢的编辑器对其中的 TimeAsm 函数进行修改，修改内容如下：

```
int TimeAsm(int argc, char *argv[])
{
    time_t tt;
    struct tm *t;
    asm volatile(
        "li a0,201\n\t"
        "ecall \n\t"
        "sd a0, %0\n\t"
        : "=m" (tt)
    );
    t = localtime(&tt);
    printf("time:%d:%d:%d:%d:%d\n", t->tm_year+1900, t->tm_mon, t->tm_mday, t->tm_hour, t->tm_min, t
        ->tm_sec);
    return 0;
}
```

另外一个需要修改的文件是 Makefile，这个文件的修改非常重要，将决定产生的什么架构的执行文件。

同样，请使用你喜欢的编辑器，打开 Makefile，将其修改为以下内容：

```
#
# Makefile for Menu Program
#
CC_PTHREAD_FLAGS    = -lpthread
CC_FLAGS            = -c
CC_OUTPUT_FLAGS     = -o
CC                  = riscv64-linux-gcc
RM                 = rm
RM_FLAGS           = -f
TARGET   = test
OBJS     = linktable.o menu.o test.o
all: $(OBJS)
```

```
$(CC) $(CC_OUTPUT_FLAGS) $(TARGET) $(OBJS)

rootfs:
riscv64-linux-gcc -o init linktable.c menu.c test.c -static -lpthread
riscv64-linux-gcc -o hello hello.c -static
find init hello | cpio -o -Hnewc | gzip -9 > ../rootfs.img
qemu-system-riscv64 -M virt \
-kernel ../linux-5.19.16/arch/riscv/boot/Image \
-initrd ../rootfs.img \
-nographic
.c.o:
$(CC) $(CC_FLAGS) $<

clean:
$(RM) $(RM_FLAGS) $(OBJS) $(TARGET) *.bak
```

需要注意的是，本次实验需要使用 `qemu-system-riscv64` 和 `qemu-system-riscv64`，请完成第三次实验后，再来做此次实验。

使用以下命令进行编译和运行：

```
make -j$(nproc)  
make rootfs -j$(nproc)
```

成功运行后，将会显示以下信息：

图 5.1: 编译和运行

5.2 CWrite 和的编写

现在我们在 menu 目录下, 使用你喜欢的编辑器修改 test.c。这里使用神之编辑器 emacs 进行修改。

```
emacs test.c
```

打开之后，添加以下内容：

CWrite 函数内容如下：

```
int CWrite(void){  
    char s[]="hello, world\n";  
    write(1,s,13);  
    return 0;  
    return 0;  
}
```

WriteAsm 函数如下：

```
int WriteAsm(void){  
    char s[] = "hello, world\n";  
  
    __asm__ volatile(  
        "li a2, 13\n"  
        "li a0, 1\n"  
        "mv a1, %[str]\n"  
        "li a7, 64\n"  
        "ecall \n"  
        :  
        : [str] "r" (s)  
    );  
    return 0;  
}
```

同样使用以下命令进行编译和运行：

```
make -j$(nproc)  
make rootfs -j$(nproc)
```

启动 MenuOS 后：

输入 help 后回车，显示如下信息：

图 5.2: MenuOS help

再次输入 write 后回车，MenuOS 输出以下信息：

```
MenuOS>>write  
write - Write hello world  
hello, world  
MenuOS>>█
```

图 5.3: MenuOS write

输入 write-asn 后回车，MenuOS 输出以下信息：

```
MenuOS>>write-asasm
write-asasm - Write hello world(asasm)
hello, world
MenuOS>>
```

图 5.4: MenuOS write-asasm

5.3 GDB 调试 sys_write 函数

使用你喜欢的编辑器在 menu 目录下，编写 start-gdb.sh，这里依旧使用 emacs。

start-gdb.sh 的内容如下：

```
#!/bin/sh
qemu-system-riscv64 -M virt \
-kernel ../linux-5.19.16/arch/riscv/boot/Image \
-initrd ../rootfs.img \
-nographic \
-s -S
```

使用以下命令赋予 start-gdb.sh 执行的权限：

```
chmod +x start-gdb.sh
```

使用以下命令启动 start-gdb.sh：

```
./start-gdb.sh
```

启动后，shell 中应当不显示任何内容，如下所示：

```
→ menu git:(master) x ./start-gdb.sh
```

图 5.5: start-gdb.sh

另开一个终端，进入 MenuOS 的目录下，使用以下命令启动 gdb-multiarch。

```
gdb-multiarch linux-5.19.16/vmlinux
```

启动之后，输入 target remote:1234 建立连接，显示如下信息说明连接建立完成：

```
(gdb) target remote:1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb)
```

图 5.6: target remote:1234

依次使用以下命令进行 sys_write 的调试：

```
b sys_write  
c  
layout split
```

最终界面将显示如下信息：

```
fs/read_write.c
656         return ret;
657     }
658
659 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
660                  size_t, count)
661 {
662     return ksys_write(fd, buf, count);
663 }

B+> 0xffffffff80165800 <ksys_write+194>    j      0xffffffff801657b4 <ksys_write+118
0xffffffff80165802 <ksys_write+196>    sd    $1,40(sp)
0xffffffff80165804 <ksys_write+198>    sd    $2,32(sp)
0xffffffff80165806 <ksys_write+200>    auipc ra,0x540
0xffffffff8016580a <ksys_write+204>    jalr -110(ra)
B+> 0xffffffff8016580e <_se_sys_write>    addi   sp,sp,-16
0xffffffff80165810 <_se_sys_write+2>    sd    $0,0(sp)
0xffffffff80165812 <_se_sys_write+4>    sd    $a,8(sp)
0xffffffff80165814 <_se_sys_write+6>    addi   $0,sp,16
0xffffffff80165816 <_se_sys_write+8>    sext.w a0,a0

remote Thread 1.1 In: se sys write
(gdb) L662 PC: 0xffffffff8016580e
```

图 5.7: 最终界面

此时，将启动 `gdb-multiarch` 的终端和启动 `./start-gdb.sh` 的终端分别分屏左右，方便查看调试过程中的程序的输出，如下图所示：

图 5.8: gdb 分屏

执行上面打断点的操作后，在右侧图中的的最后一行显示：[0.507884] Run /init as init process 说明 Linux 内核已经初始化完成。现在使用十一次 c 命令，使得 MenuOS 加载到 shell，如下图所示：

The screenshot shows the GDB interface with two panes. The left pane displays assembly code from `fs/read_write.c`, specifically the `sys_write` function. The right pane shows the kernel's boot log. Key entries in the log include:

- [0.28531] pci-host generic 38000000 pci_ECAN_1 [mem 0x30000000-0x3fffffff] for [bus 00-ff]
- [0.29710] pci bus 0000:00 root bus resource [bus 00-ff]
- [0.29716] pci bus 0000:00 root bus resource [io 0x0000-0xffff]
- [0.29717] pci bus 0000:00 root bus resource [mem 0x00000000-0xffffffff]
- [0.29718] pci bus 0000:00 root bus resource [mem 0x40000000-0xffffffff]
- [0.29719] Freeing untrd memory: 632K
- [0.36230] Serial: 8259/16559 driver, 4 ports, I/O sharing disabled
- [0.36231] E1000E Intel(R) PRO/1000 MT Desktop [mem 0x00000000-0xffffffff] (irq = 1, base_baud = 230400) is a 16559A
- [0.39703] printk: console [ttyS0] enabled
- [0.41907] e1000e: Intel(R) PRO/1000 MT Desktop [mem 0x00000000-0xffffffff] (irq = 1, base_baud = 230400)
- [0.41907] e1000e: Copyright 1999 - 2013 Intel Corporation.
- [0.41932] e1000e: PCI Express gigabit Ethernet driver
- [0.42625] ehci-pci: EHCI PCI platform driver
- [0.42626] ohci-hcd: OHCI 1.1 host controller (OHCI) Driver
- [0.42113] ohci-pci: OHCI PCI platform driver
- [0.42317] usbcore: registered new interface driver us
- [0.42306] usbcore: registered new interface driver storage
- [0.42276] goldfish_rtc: registered as rtc0
- [0.42276] goldfish_rtc: registered as rtc1
- [0.43001] syscon-poweroff socpowernf: pm_power off already claimed for sbi_srst power off
- [0.43001] syscon-poweroff socpowernf: pm_power off already claimed for sbi_srst power off
- [0.43317] sdhci: SDHCI generic driver
- [0.43317] sdhci: Digital SD/MMC Interface driver
- [0.43379] sdhci1: Pierre Ossman's SDHCI driver helper
- [0.43556] usbcore: registered new interface driver ushid
- [0.43602] ushid: USB HID core driver
- [0.43602] ushid: Usbhid extension is available
- [0.43712] riscv-pmu-sb1: 15 Firmware and 18 hardware counters
- [0.43712] riscv-pmu-sb1: 15 Firmware and 18 hardware counters
- [0.44205] NET: Registered PF_INET6 protocol family
- [0.456129] Segment Routing with IPv6
- [0.456129] Segment Routing with IPv6
- [0.451322] SIT: IPv6, IPv4 and MPLS over IP4 tunneling driver
- [0.455281] 9pnet: 9P2000 protocol family
- [0.455281] 9pnet: Installing SP2008 support
- [0.456118] Key type dns_resolver registered
- [0.456118] Key type dns_resolver registered
- [0.49770] Freeing unused kernel image (inittime) memory: 2172K
- [0.36540] Run /init as init process

图 5.9: MenuOS 加载 shell

现在在 MenuOS 中输入 write-asn 回车之后，MenuOS 将会暂停输出。然后在左侧 gdb 窗口中输入命令 c，MenuOS 将会显示以下信息：

The terminal window shows the output of the `write-asn` command. The output consists of a grid of asterisks (*), followed by the text "MenuOS>>write-asn" and "write-asn - Write hello world(asn)".

图 5.10: MenuOS 运行 write-asn

此时可以开始分析 Cwrite 中使用系统调用 write 的过程。首先使用以下命令对 handle_exception 打断点：

b handle_exception

然后使用命令 c 进入到 handle_exception 断点处。在 handle_exception 函数中，根据异常号判断是否是系统调用。如果是系统调用，控制权会传递到系统调用处理的分发函数。

The assembly view shows the `handle_exception` function. The code is as follows:

```

* @regs:      Register file coming from the low-level handling code
*/
asm linkage void noinstr generic_handle_arch_irq(struct pt_regs *regs)
{
    struct pt_regs *old_regs;

    irq_enter();
    old_regs = set_irq_regs(regs);
    handle_arch_irq(regs);
    set_irq_regs(old_regs);
    irq_exit();
}
#endif

```

图 5.11: handle_exception 断点处

generic_handle_arch_irq 是 Linux 内核中的一个通用中断处理函数，它用于处理系统中断的基本工作。这个函数的主要作用是将控制权传递给适当的中断处理函数。在 Linux 内核中，每个中断都有一个对应的中断处理函数，用于处理特定类型的中断。generic_handle_arch_irq 通过查询中断描述符表（Interrupt Descriptor Table，IDT）来确定要执行的中断处理函数，并将控制权转交给它。

使用命令 c 进入 ksys_write 断点处。在 SYSCALL_DEFINE3 的函数中的返回语句 return ksys_write(fd, buf, count);，如下图所示：

```

fs/read_write.c
655     return ret;
656 }
658
659 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
660                  size_t, count)
661 {
662     return ksys_write(fd, buf, count);
663 }
664
665 ssize_t ksys_pread64(unsigned int fd, char __user *buf, size_t count,
666                      loff_t pos)
667 {
668     struct fd f;
669     ssize_t ret = -EBADF;
670

```

图 5.12: ksys_write 断点处

继续使用 si 命令，将程序将执行下一条汇编语句，如下图所示：

```

B+ 0xffffffff8016580e <_se sys write>    addi   sp,sp,-16
0xffffffff80165810 < se sys write+2>    sd    $0,0(sp)
> 0xffffffff80165812 < se sys write+4>    sd    ra,0(sp)
0xffffffff80165814 < se sys write+6>    addi   $0,sp,16
0xffffffff80165816 < se sys write+8>    sext.w a0,a0
0xffffffff80165818 < se sys write+10>   jal    ra,0xffffffff8016573e <ksys_write>
0xffffffff8016581c < se sys write+14>   ld    ra,8(sp)
0xffffffff8016581e < se sys write+16>   ld    $0,0(sp)
0xffffffff80165820 < se sys write+18>   addi   sp,sp,16
0xffffffff80165822 < se sys write+20>   ret
0xffffffff80165824 <ksys_pread64>      addi   sp,sp,-80
0xffffffff80165826 <ksys_pread64+2>    sd    $0,64(sp)
0xffffffff80165828 <ksys_pread64+4>    sd    ra,72(sp)
0xffffffff8016582a <ksys_pread64+6>    addi   $0,sp,80
0xffffffff8016582c <ksys_pread64+8>    sd    s3,40(sp)
0xffffffff8016582e <ksys_pread64+10>   sd    a3,-72($0)
0xffffffff80165832 <ksys_pread64+14>   bitz  a3,0xffffffff801658b0 <ksys_pread64+140>

```

图 5.13: 执行下一条汇编语句

system_call 调用过程如下：

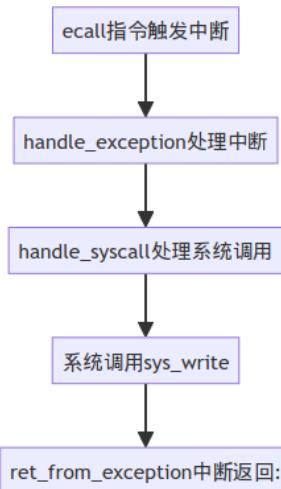


图 5.14: system_call 调用过程

具体的流程由于一些限制，光靠这个 gdb 无法明确的知道系统调用的一些细节，在这里展开讲解一些。

ecall 触发 trap，ecall 它其实是跳转到一个地址，这个地址被存在 stvec 寄存器中（请了解 RISCV 特权架构指令）这个 stvec 的地址在操作系统的启动的时候就已经被设置好了在 arch/riscv/kernel/head.S 里面，它就是 handle_exception。handle_exception 会保存用户的上下文，然后把内核上下文写入寄存器（包括页表）sstatus、sepc、stval、scause、sscratch 这 5 个 csr 寄存器（请了解 RISCV 特权架构指令）存储了触发 trap 的信息。scause 里面存储的是出发 trap 的信息，scause 寄存器最高位含义如下：最高位 =1: interrupts 最高位 =0: exceptions，如 scause 的值大于 0 那么就是由 exceptions 触发，如果小于 0，那么就是一个中断。我们这里是大于 0 的。

接下来判断 scause 是否等于 EXC_SYSCALL 这个值是 8，也就是当 scause==8 时，就表示由 sys_call 触发的 trap，从而进入 handle_syscall handle_syscall 里面会将 spec 也就是出错的地址（我们这里是 ecall 的地址）+4 并且存储下来，用于正确返回（sret 的时候是 ret 到下一个 ecall 的地址防止无限循环），然后根据 a7 里面的值确定是哪个系统调用。找到相应的系统调用，返回。注意在 handle_syscall 里跳转到实际的系统调用函数时把返回地址设置成了 ret_from_syscall。所以上述的 sys_write 函数返回后会跳转到 ret_from_syscall 继续执行。ret_from_syscall 会将保存的用户模式的上下文中的 a0（返回值）切换为系统调用运行之后的返回值，然后恢复现场，最后使用 sret 来回到用户态

第 6 章 实验六：分析 Linux 内核创建一个新进程的过程

实验要求：

- 阅读理解 task_struct 数据结构；
- 分析 fork 函数对应的内核处理过程 sys_clone，理解创建一个新进程如何创建和修改 task_struct 数据结构；
- 使用 gdb 跟踪分析一个 fork 系统调用内核处理函数 sys_clone
- 验证对 Linux 系统创建一个新进程的理解。特别关注新进程是从哪里开始执行的？为什么从那里能顺利执行下去？即执行起点与内核堆栈如何保证一致。

6.1 阅读理解 task_struct 数据结构

struct task_struct 结构体是用于描述进程的结构体。它非常庞大，其中列举几个常用的结构体成员以及用途：

- thread_info 描述进程的底层信息
- mm_struct 指向内存区域描述符的指针
- tty_struct 描述进程相关的 tty 设备
- pid 是进程的标识符
- fs_struct 表示当前目录
- state 是进程状态
- files_struct 指向文件描述符的指针
- stack 是堆栈
- thread 用于保存上下文中 CPU 相关状态信息的关键数据
- signal_struct 表示接收到的信号

6.2 分析 fork 函数对应的内核处理过程 sys_clone

对于 RISC-V Linux 系统来说，用户态程序执行 ecall 指令出发 entry_SYSCALL64 并以 ret 返回系统调用。系统调用陷入内核态，用用户态堆栈转换到内核态堆栈。

fork 系统调用创建了一个子进程，子进程复制了父进程中所有的信息，包括内核堆栈、进程描述符等。fork 系统调用在内核里面变成了父、子两个进程，父进程正常 fork 系统调用返回用户态，子进程也要从内核态返回用户态。

在 RISC-V Linux 系统中没有定义 fork 系统调用，只在 include/uapi/asm-generic/unistd.h 定义了 clone 系统调用，对应的内核处理函数为 220 号系统调用 sys_clone。

```
636 #define __NR_keyctl 219
637 __SC_COMP(__NR_keyctl, sys_keyctl, compat_sys_ke
638
639 /* arch/example/kernel/sys_example.c */
640 #define __NR_clone 220
641 __SYSCALL(__NR_clone, sys_clone)
642 #define __NR_execve 221
643 __SC_COMP(__NR_execve, sys_execve, compat_sys_ex
644 #define __NR3264_mmap 222
```

图 6.1: 系统调用 sys_clone

在 Linux 5.19.16 中这些创建进程的系统调用都是在 kernel/fork 文件中实现的，代码如下：

```

pid_t kernel_clone(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p; //待创建进程的进程描述符
    int trace = 0;
    pid_t nr; //待创建进程的pid

    if ((args->flags & CLONE_PIDFD) &&
        (args->flags & CLONE_PARENT_SETTID) &&
        (args->pidfd == args->parent_tid))
        return -EINVAL;

    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if (args->exit_signal != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }

    //复制进程描述符和执行时所需的其他数据结构
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    add_latent_entropy();

    if (IS_ERR(p))
        return PTR_ERR(p);

    trace_sched_process_fork(current, p);

    pid = get_task_pid(p, PIDTYPE_PID);
    nr = pid_vnr(pid);

    if (clone_flags & CLONE_PARENT_SETTID)
        put_user(nr, args->parent_tid);

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }

    wake_up_new_task(p); //将子进程添加到就绪队列，使之有机会被调度执行
}

```

```

/* forking complete and child started to run, tell ptracer */
if (unlikely(trace))
    ptrace_event_pid(trace, pid);

if (clone_flags & CLONE_VFORK) {
    if (!wait_for_vfork_done(p, &vfork))
        ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
}

put_pid(pid);
return nr;
}

pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    struct kernel_clone_args args = {
        .flags   = ((lower_32_bits(flags) | CLONE_VM |
                    CLONE_UNTRACED) & ~CSIGNAL),
        .exit_signal = (lower_32_bits(flags) & CSIGNAL),
        .fn     = fn,
        .fn_arg  = arg,
        .kthread = 1,
    };

    return kernel_clone(&args);
}

pid_t user_mode_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    struct kernel_clone_args args = {
        .flags   = ((lower_32_bits(flags) | CLONE_VM |
                    CLONE_UNTRACED) & ~CSIGNAL),
        .exit_signal = (lower_32_bits(flags) & CSIGNAL),
        .fn     = fn,
        .fn_arg  = arg,
    };

    return kernel_clone(&args);
}

#endif __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    struct kernel_clone_args args = {
        .exit_signal = SIGCHLD,

```

```

};

return kernel_clone(&args);
#else
/* can not support in nommu mode */
return -EINVAL;
#endif
}

#endif

#endif /* __ARCH_WANT_SYS_VFORK */

SYSCALL_DEFINE0(vfork)
{
    struct kernel_clone_args args = {
        .flags    = CLONE_VFORK | CLONE_VM,
        .exit_signal = SIGCHLD,
    };

    return kernel_clone(&args);
}
#endif

#endif /* __ARCH_WANT_SYS_CLONE */

#ifndef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
               int __user *, parent_tidptr,
               unsigned long, tls,
               int __user *, child_tidptr)
#endif
#endif /* CONFIG_CLONE_BACKWARDS2 */

```

从上面的代码可以看出，fork、vfork 和 clone 这三个系统调用以及 user_mode_thread 和 kernel_thread 内核函数都能创建一个新的进程，而且都是通过 kernel_clone 这个函数来创建的，只是传递的参数不一样。

理解创建一个新进程如何创建和修改 task_struct 数据结构创建一个进程是复制当前进程的信息，通过 kernel_clone 函数创建一个新的进程。因为父进程和子进程的绝大部分信息是一样的，除开 pid 的值和内核堆栈以及 thread 结构体变量记录进程执行上下文的 CPU 关键信息。

kernel_clone 的代码如下：

```

pid_t kernel_clone(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p; //待创建进程的进程描述符
    int trace = 0;
    pid_t nr; //待创建进程的pid

    if ((args->flags & CLONE_PIDFD) &&
        (args->flags & CLONE_PARENT_SETTID) &&
        (args->pidfd == args->parent_tid))

```

```

return -EINVAL;

if (!(clone_flags & CLONE_UNTRACED)) {
    if (clone_flags & CLONE_VFORK)
        trace = PTRACE_EVENT_VFORK;
    else if (args->exit_signal != SIGCHLD)
        trace = PTRACE_EVENT_CLONE;
    else
        trace = PTRACE_EVENT_FORK;

    if (likely(!ptrace_event_enabled(current, trace)))
        trace = 0;
}

//复制进程描述符和执行时所需的其他数据结构
p = copy_process(NULL, trace, NUMA_NO_NODE, args);
add_latent_entropy();

if (IS_ERR(p))
    return PTR_ERR(p);

trace_sched_process_fork(current, p);

pid = get_task_pid(p, PIDTYPE_PID);
nr = pid_vnr(pid);

if (clone_flags & CLONE_PARENT_SETTID)
    put_user(nr, args->parent_tid);

if (clone_flags & CLONE_VFORK) {
    p->vfork_done = &vfork;
    init_completion(&vfork);
    get_task_struct(p);
}

wake_up_new_task(p); //将子进程添加到就绪队列，使之有机会被调度执行

/* forking complete and child started to run, tell ptracer */
if (unlikely(trace))
    ptrace_event_pid(trace, pid);

if (clone_flags & CLONE_VFORK) {
    if (!wait_for_vfork_done(p, &vfork))
        ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
}
put_pid(pid);
return nr;
}

```

从这段代码可以看出创建一个新进程的流程如下：

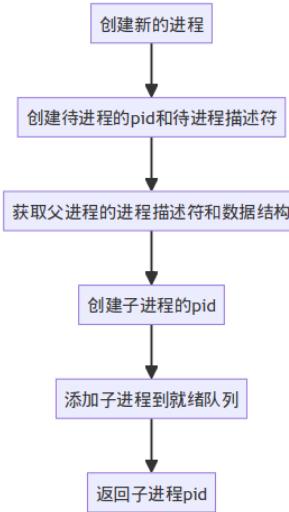


图 6.2: 创建一个新进程

而对于创建一个新进程进而修改 task_struct 数据结构的秘密就在获取父进程的进程描述符和其他数据结构的步骤之中，涉及的函数就是 copy_process()。限于篇幅，现在简短的讲解 copy_process() 的内容。

copy_process() 函数主要是做了以下几件事情：

1. 复制进程描述符 task_struct、创建内核堆栈
2. 复制所有的进程信息
3. 初始化子进程和 thread
4. 为子进程分配新的 pid
5. 增加系统中的进程数目
6. 返回被创建的子进程描述符指针

而第一件事情——复制进程描述符 task_struct、创建内核堆栈，调用了 dup_task_struct 函数，而这个函数是实际复制进程描述符的关键函数。

6.3 GDB 跟踪分析 sys_clone

使用 gdb 跟踪分析一个 fork 系统调用内核处理函数 sys_clone，我们在 menuos 中的 test.c 文件中的添加了 ForkProcess 函数，内容如下：

```

int ForkProcess(int argc, char *argv[]){
    int pid;
    pid = fork();
    if(pid < 0){
        fprintf(stderr, "Fork Failed!");
        exit(-1);
    }
    else if (pid == 0){
        printf("This is Child Process!\n");
    }
    else {
        printf("This is Parent Process!\n");
        wait(NULL);
    }
}
  
```

```

    printf("Child Complete!\n");
}
}

```

并在 main 函数中添加以下语句：

```
MenuConfig("fork-new", "fork new process", ForkProcess);
```

在 menuos 的目录下使用以下命令启动 qemu 模拟 riscv 架构同时启动 gdbserver：

```
./init-gdb.sh
```

新建另一个 shell 在 menuos 目录中输入以下命令，启动 gdb-multiarch 并加载 Linux 内核符号表：

```
./start-gdb.sh
```

然后依次打入 sys_clone 、 kernel_clone 、 dup_task_struct 、 copy_process 、 copy_thread 、 ret_from_fork 的断点。

```

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ../linux-5.19.16/vmlinux...
(gdb) target remote:1234
Remote debugging using :1234
0x000000000001000 in ?? ()
(gdb) b sys_clone
Breakpoint 1 at 0xffffffff8000d568: file kernel/fork.c, line 2758.
(gdb) b do_fork
Function "do_fork" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) Quit
(gdb) b kernel_clone
Breakpoint 2 at 0xffffffff8000cfaa: file kernel/fork.c, line 2616.
(gdb) b dup_task_struct
Breakpoint 3 at 0xffffffff8000bf48: file kernel/fork.c, line 967.
(gdb) b copy_process
Breakpoint 4 at 0xffffffff8000bdcc: file kernel/fork.c, line 1981.
(gdb) b copy_thread
Breakpoint 5 at 0xffffffff80003834: file arch/riscv/kernel/process.c, line 162.
(gdb) b ret_from_fork
Breakpoint 6 at 0xffffffff80003504
(gdb) []

```

图 6.3: 打入断点

从上图中可以发现 do_fork 这个函数的确已经在 5.19.16 的内核删除，并使用 kernel_clone 替代了。

待程序成功加载 Menuos 后，在 MenuOS 中输入 fork-new 使用 ForkProcess 函数创建新的进程。

```
* *
*** * *    **    **    *    *    *    *    ****    ****
* * * *    * *    * *    *    *    *    *    **
* * * *    * *    * *    *    *    *    *    ***    *
* *** *    *****    *    *    *    *    *    *    **
*    *    *    *    *    *    *    *    *    *    **
*    *    *    *    *    *    *    *    *    *    **
*    *    *    *    *    *    *    *    *    *    **
*    *    ***    *    *    *    *    *    ****    ****

MenuOS>>help
help - Menu List
    * help - Menu List
    * version - MenuOs V2.0(Based on Linux 5.19.16)
    * quit - Quit from MenuOS
    * time - Show System Time
    * time-asm - Show System Time(asm)
    * write - Write hello world
    * write-asm - Write hello world(asm)
    * fork-new - fork new process
MenuOS>>forl-mew
MenuOS>>fork-new
fork-new - fork new process
[]
```

图 6.4: fork-new

首先启动的是 __se_sys_clone, 如下图所示:

```
Breakpoint 1, __se_sys_clone (clone_flags=18874385, newsp=0, parent_tidptr=0, tls=0, child_tidptr=671952) at kernel/fork.c:2758
2758      SYSCALL_DEFINES(clone, unsigned long, clone_flags, unsigned long, newsp,
(gdb) c
Continuing.
```

图 6.5: __se_sys_clone

然后启动的是 kernel_clone:

```
Breakpoint 2, kernel_clone (args=args@entry=0xff2000000060be08) at kernel/fork.c:2616
2616 {
(gdb) c
Continuing.
```

图 6.6: se_sys_clone

接着是 copy_process:

```
Breakpoint 4, copy_process (pid=pid@entry=0x0, trace=trace@entry=0, node=node@entry=-1, args=args@entry=0xff2000000060be08) at kernel/fork.c:1981
(gdb) l
1976     static __latent_entropy struct task_struct *copy_process(
1977             struct pid *pid,
1978             int trace,
1979             int node,
1980             struct kernel_clone_args *args)
1981 {
1982     int pidfd = -1, retval;
1983     struct task_struct *p;
1984     struct multiprocess_signals delayed;
1985     struct file *pidfile = NULL;
(gdb) c
Continuing.
```

图 6.7: copy_process

然后开始复制所有的进程信息

```
(gdb) c
Continuing.

Breakpoint 3, dup_task_struct (node=-1, orig=0xff6000001638000) at kernel/fork.c:967
967         if (node == NUMA_NO_NODE)
(gdb) l
962     static struct task_struct *dup_task_struct(struct task_struct *orig, int node)
963 {
964     struct task_struct *tsk;
965     int err;
966
967     if (node == NUMA_NO_NODE)
968         node = tsk_fork_get_node(orig);
969     tsk = alloc_task_struct_node(node);
970     if (!tsk)
971         return NULL;
(gdb) []
```

图 6.8: 复制所有的进程信息

接着进入 copy_thread

```
(gdb) 
Continuing.

Breakpoint 5, copy_thread (p=p@entry=0xff60000002090000, args=args@entry=0xff2000000060be08) at arch/riscv/kernel/process.c:162
162         unsigned long clone_flags = args->flags;
(gdb) l
157     return 0;
158 }
159
160 int copy_thread(struct task_struct *p, const struct kernel_clone_args *args)
161 {
162     unsigned long clone_flags = args->flags;
163     unsigned long usp = args->stack;
164     unsigned long tls = args->tls;
165     struct pt_regs *childregs = task_pt_regs(p);
166
(gdb) []
```

图 6.9: copy_thread

最后返回子进程

```

166
(gdb) c
Continuing.

Breakpoint 6, 0xffffffff80003504 in ret_from_fork ()
(gdb) l
167          /* p->thread holds context to be restored by __switch_to() */
168          if (unlikely(args->fn)) {
169              /* Kernel thread */
170              memset(childregs, 0, sizeof(struct pt_regs));
171              childregs->gp = gp_in_global;
172              /* Supervisor/Machine, irqs on: */
173              childregs->status = SR_PP | SR_PIE;
174
175          p->thread.ra = (unsigned long)ret_from_kernel_thread;
176          p->thread.s[0] = (unsigned long)args->fn;
(gdb) []

```

图 6.10: 返回子进程

MenuOS 上如下：

```

MenuOS>>help
help - Menu List
  * help - Menu List
  * version - MenuOS V2.0(Based on Linux 5.19.16)
  * quit - Quit from MenuOS
  * time - Show System Time
  * time-asm - Show System Time(asm)
  * write - Write hello world
  * write-asm - Write hello world(asm)
  * fork-new - fork new process
MenuOS>>fork-new
MenuOS>>fork-new
fork-new - fork new process
This is Parent Process!
This is Child Process!
MenuOS>>[]

```

图 6.11: MenuOS

本次调试过程基本符合之前分析的步骤。

第7章 实验七：Linux 内核如何装载和启动一个可执行程

实验要求：

- 理解编译链接的过程和 ELF 可执行文件格式
- 编程使用 exec 库函数加载一个可执行文件，动态链接分为可执行程序装载时动态链接和运行时动态链接，编程练习动态链接库的这两种使用方式
- 使用 gdb 跟踪分析一个 execve 系统调用内核处理函数 sys_execve
- 验证对 Linux 系统加载可执行程序所需处理过程的理解。特别关注新的可执行程序是从哪里开始执行的？
- 为什么 execve 系统调用返回后新的可执行程序能顺利执行？对于静态链接的可执行程序和动态链接的可执行程序 execve 系统调用返回时会有什么不同？

7.1 程序的编译过程

程序从源代码到可执行文件的编译步骤分为 4 步：

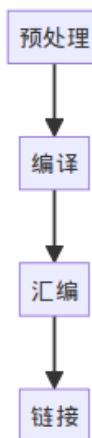


图 7.1：编译过程

在预处理过程中，主要是完成删除和展开以及处理命令这三种操作，最后将文本保存到后缀为.i 文件中。以预处理 hello.c 文件为例。

下面是 hello.c 文件的内容：

```
#include <stdio.h>

void main(){
    printf("hello world!\n");
}
```

使用 ssh 连接好 visionfive 2 开发板后，使用 vim 创建 hello.c 文件。

然后执行以下的预处理命令得到 hello.i 文件：

```
gcc -E hello.c -o hello.i
```

使用 vim 打开 hello.i 文件后内容较多，但可以看出有几大特点：

首先是 #include 命令全部替换成包含文件的路径，如下图所示：

```

1 # 0 "hello.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "hello.c"
7 # 1 "/usr/include/stdio.h" 1 3 4
8 # 27 "/usr/include/stdio.h" 3 4
9 # 1 "/usr/include/riscv64-linux-gnu/bits/libc-header-start.h" 1 3 4
10 # 33 "/usr/include/riscv64-linux-gnu/bits/libc-header-start.h" 3 4
11 # 1 "/usr/include/features.h" 1 3 4
12 # 392 "/usr/include/features.h" 3 4
13 # 1 "/usr/include/features-time64.h" 1 3 4
14 # 20 "/usr/include/features-time64.h" 3 4
15 # 1 "/usr/include/riscv64-linux-gnu/bits/wordsize.h" 1 3 4
16 # 21 "/usr/include/features-time64.h" 2 3 4
17 # 1 "/usr/include/riscv64-linux-gnu/bits/timesize.h" 1 3 4
18 # 22 "/usr/include/features-time64.h" 2 3 4
19 # 393 "/usr/include/features.h" 2 3 4
20 # 489 "/usr/include/features.h" 3 4
21 # 1 "/usr/include/riscv64-linux-gnu/sys/cdefs.h" 1 3 4
22 # 559 "/usr/include/riscv64-linux-gnu/sys/cdefs.h" 3 4
23 # 1 "/usr/include/riscv64-linux-gnu/bits/wordsize.h" 1 3 4
24 # 560 "/usr/include/riscv64-linux-gnu/sys/cdefs.h" 2 3 4
25 # 1 "/usr/include/riscv64-linux-gnu/bits/long-double.h" 1 3 4
26 # 561 "/usr/include/riscv64-linux-gnu/sys/cdefs.h" 2 3 4
27 # 490 "/usr/include/features.h" 2 3 4
28 # 513 "/usr/include/features.h" 3 4
29 # 1 "/usr/include/riscv64-linux-gnu/gnu/stubs.h" 1 3 4

```

图 7.2: #include 替换

同时添加了行号和文件名标示：

```

41
42
43 # 3 "hello.c"
44 void main()[
45     printf("Hello world!\n");
46 ]

```

图 7.3: 添加标示

其次是删除了所有的注释。

编译是在预处理的基础上，gcc 首先检查代码的规范性和语法错误，检查无误后的翻译为汇编语言。

使用以下的命令将预处理的中的代码生成为汇编语言代码：

```
gcc -S hello.i -o hello.s
```

以下的在 visionfive 2 中编译完的 RISCV 架构的汇编代码：

```

.file "hello.c"
.option pic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section .rodata
.align 3
.LCO:
.string "Hello world!"
.text

```

```

.align 1
.globl main
.type main, @function
main:
addi sp,sp,-16
sd ra,8(sp)
sd s0,0(sp)
addi s0,sp,16
lla a0,.LC0
call puts@plt
nop
ld ra,8(sp)
ld s0,0(sp)
addi sp,sp,16
jr ra
.size main, .-main
.ident "GCC: (Debian 12.2.0-10) 12.2.0"
.section .note.GNU-stack,"",@progbits

```

链接就是将各个代码和数据收集起来组合成一个单一文件的过程。以下的命令将生成 RISC-V Linux 下的可执行文件：

```
gcc hello.o -o hello -static
```

以下是节表头：

```

There are 28 section headers, starting at offset 0x7a490:

Section Headers:
[Nr] Name          Type        Address      Offset
Size       EntSize     Flags Link Info Align
[ 0]             NULL        0000000000000000 00000000
0000000000000000 0000000000000000      0   0   0
[ 1] .note.gnu.bu [...] NOTE      0000000000101c8 000001c8
0000000000000024 0000000000000000 A   0   0   4
[ 2] .note.ABI-tag NOTE      0000000000101ec 000001ec
0000000000000020 0000000000000000 A   0   0   4
[ 3] .rela.dyn    RELA      000000000010210 00000210
00000000000000210 0000000000000018 A   25  0   8
[ 4] .text         PROGBITS  000000000010420 00000420
00000000000412b2 0000000000000000 AX  0   0   4
[ 5] __libc_free_fn PROGBITS  00000000000516d2 000416d2
0000000000000814 0000000000000000 AX  0   0   2
[ 6] .rodata       PROGBITS  0000000000051ef0 00041ef0
000000000001b4a4 0000000000000000 A   0   0   16

```

编程使用 exec 库函数加载一个可执行文件在 visionfive 2 开发板上启动 debain 后，使用以下的命令使用 man 命令查看 execve 函数信息：

```
man execve
```

execve 函数信息如下：

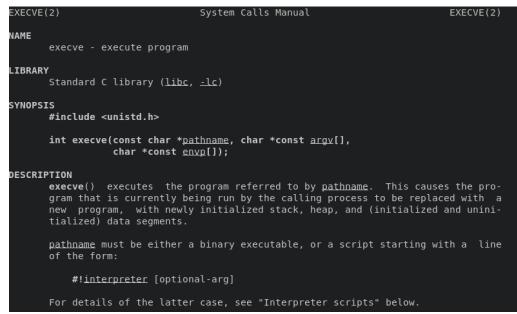


图 7.4: execve 函数信息

现在编写一个叫作 local_exec.c 的文件，使用 exec 系统调用加载之前编译产生的 hello 可执行文件。下面是 local_exec.c 的内容：

```

#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0)
    {
        /* error occurred */
        fprintf(stderr,"Fork Failed!");
        exit(-1);
    }
    else if (pid == 0)
    {
        /* child process */
        printf("This is Child Process!\n");
        execlp("/hello","hello",NULL);
    }
    else
    {
        /* parent process */
        printf("This is Parent Process!\n");
        /* parent will wait for the child to complete*/
        wait(NULL);
        printf("Child Complete!\n");
    }
}

```

在 exec 函数中，第一个参数是可执行文件的路径，第二个参数是程序的名称，最后一个参数必须是 (char *)0。
使用以下命令进行编译：

```
gcc -o local_exec local_exec.c
```

运行结果如下：

```
root@starfive:~/Code# ./local_exec
This is Parent Process!
This is Child Process!
Child Complete!
```

图 7.5: local_exec

7.2 动态链接

动态链接是一种在程序运行时将代码和数据与程序链接的技术，而不是在编译时将它们链接到可执行文件中。它提供了灵活性和共享性，允许多个程序共享和重复使用相同的库代码。动态链接主要分为两种方式：可执行程序装载时动态链接和运行时动态链接。相比于静态链接，动态链接可以多个程序共享同一个段代码，而不需要多个副本。但是动态链接对库的依赖程度高。

可执行程序装载时动态链接是指动态链接发生在可执行程序被加载到内存时查找和链接动态链接库。运行时动态链接是指程序在运行时显式加载并链接动态链接库，然后调用库中的函数。

可执行程序装载时动态链接通常更加高效，因为它减少了运行时的开销，而运行时动态链接提供了更大的灵活性，允许动态加载和卸载库，以便在运行时进行插件或模块式的扩展。

gcc 默认编译就是动态链接，使用以下命令动态链接 hello 可执行文件：

```
gcc hello.o -o hello.dynamic
```

使用 ls -l 命令可以发现在 RISC-V 架构下动态链接的可执行文件比静态链接的可执行文件小 58 倍，而书上比较 x86 的比值是 100 倍：

```
root@starfive:~/Code# ls -l
total 536
-rwxr-xr-x 1 root root 502672 Nov  8 12:34 hello
-rw-r--r-- 1 root root     61 Nov  8 07:08 hello.c
-rwxr-xr-x 1 root root  8560 Nov  8 13:05 hello.dynamic
-rw-r--r-- 1 root root 17933 Nov  8 07:18 hello.i
-rw-r--r-- 1 root root  1680 Nov  8 12:31 hello.o
-rw-r--r-- 1 root root    527 Nov  8 12:27 hello.s
```

图 7.6: 文件大小比较

以下是用于实践动态链接的代码：

首先是可执行程序装载时动态链接代码，我们需要创建两个文件，分别是 shlibexample.h 和 shlibexample.c 文件。

shlibexample.h 源代码如下：

```
#ifndef _SH_LIB_EXAMPLE_H_
#define _SH_LIB_EXAMPLE_H_

#define SUCCESS 0
#define FAILURE (-1)

#ifndef __cplusplus
extern "C"{
#endif

int SharedLibApi();
```

```
}
#endif
#endif
```

shlibexample.c 源代码如下：

```
#include <stdio.h>
#include "shlibexample.h"

int SharedLibApi(){
    printf("This is a shared library!\n");
    return SUCCESS;
}
```

保存好后文件后，在 visionfive 2 开发板使用以下命令，创建 libshlibexample.so 文件：

```
gcc -shared shlibexample.c -o libshlibexample.so
```

实现运行时动态链接，我们也需要创建两个文件，分别是 dllibexample.h 和 dllibexample.c 文件。

dllibexample.h 源代码如下：

```
#ifndef _DL_LIB_EXAMPLE_H_
#define _DL_LIB_EXAMPLE_H_

#ifndef __cplusplus
extern "C"{
#endif

int DynamicalLoadingLibAPi();

#ifndef __cplusplus
}
#endif
#endif
#endif
```

dllibexample.c 源代码如下：

```
#include <stdio.h>
#include "dllibexample.h"

#define SUCCESS 0
#define FAILURE (-1)

int DynamicalLoadingLibAPi(){
    printf("This is a Dynamical Loading library!\n");
    return SUCCESS;
}
```

同样保存好后文件后，在 visionfive 2 开发板使用以下命令，创建 libdllibexample.so 文件：

```
gcc -shared dllibexample.c -o libdllibexample.so
```

当分别生成好 libshlibexample.so 文件和 libdllibexample.so 文件后，我们需要分别调用这两个文件。在 visionfive 2 开发板中创建一个名为 main.c 的文件，main.c 的内容如下：

```
#include <stdio.h>
#include "shlibexample.h"
#include <dlfcn.h>

int main(){
    printf("Calling SharedLibApi() function of libshlibexample.so!\n");
    SharedLibApi();

    void * handle = dlopen("libdllibexample.so", RTLD_NOW);
    if(handle == NULL){
        printf("Open Lib libdllibexample.so Error:%s\n", dlerror());
        return FAILURE;
    }
    int (*func)(void);
    char *error;
    func = dlsym(handle , "DynamicalLoadingLibAPi");
    if((error = dlerror()) != NULL){
        printf("DynamicalLoadingLibAPi() not found:%s\n", error);
        return FAILURE;
    }
    printf("Calling DynamicalLoadingLibApi() function of libdllibexample.so!\n");
    func();
    dlclose(handle);
    return SUCCESS;
}
```

编辑好保存后，使用以下命令编译成 main 可执行文件：

```
gcc main.c -o main -L/root/Code/ -lshlibexample -ldl
```

然后将之前生成的 shlibexample.so 文件拷贝到 /usr/local/lib 中否则程序无法找到 shlibexample.so 文件。完成以上工作后，使用以下命令，即可实现可执行程序装载时动态链接和运行时动态链接：

```
./main
```

成功运行后，会显示以下信息：

```
root@starfive:~/Code# ./main
Calling SharedLibApi() function of libshlibexample.so!
This is a shared library!
Calling DynamicalLoadingLibApi() function of libdllibexample.so!
This is a Dynamical Loading library!
root@starfive:~/Code# []
```

图 7.7: 可执行程序装载时动态链接和运行时动态链接

7.3 gdb 跟踪分析一个 execve 系统调用

进入 MenuOS 目录中的 menuos 目录后，编辑 test.c，在其中加入以下代码：

首先添加以下的头文件

```
#include <unistd.h>
```

然后添加 Exec 函数:

```
int Exec(int argc, char *argv[])
{
    int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0)
    {
        /* error occurred */
        fprintf(stderr,"Fork Failed!");
        exit(-1);
    }
    else if (pid == 0)
    {
        /* child process */
        printf("This is Child Process!\n");
        execlp("/hello","hello",NULL);
    }
    else
    {
        /* parent process */
        printf("This is Parent Process!\n");
        /* parent will wait for the child to complete*/
        wait(NULL);
        printf("Child Complete!\n");
    }
}
```

最后在 main 函数中添加以下语句:

```
MenuConfig("execve", "execve new process", Exec);
```

在 menus 目录下使用以下语句进行编译:

```
make
make rootfs
```

编译完成后，在两个终端上分别启动 init-gdb.sh 和 start-gdb.sh 脚本。脚本启动后，在启动 start-gdb.sh 的终端上输入 target remote:1234 连接 qemu 中的 gdbserver。完成后，使用下述命令在 gdb 中为 sys_execve 设置断点:

```
b sys_execve
```

打好断点后，使用 c 命令，加载 MenuOS 后输入 exec 开始调试 exec，如下图所示：

```

0.378028] sdhci-pifm: SDHCI platform and OF driver helper
0.378028] sdhci-pifm: using direct memory interface driver ushid
0.378028] ushid: USB HCD core driver
0.379735] riscv-pmu-sbi: SBI PMU extension is available
0.380279] riscv-pmu-sbi: 15 firmware and 18 hardware counters
0.380279] riscv-pmu-sbi: Perf sampling/filtering is not supported as sscof extension is not available
0.383099] NET: Registered PF_INET6 protocol family
0.390132] Segment Routing with IPv6
0.390132] DVB API device driver
0.390911] sit: IPv6 Ingress IP4 and MPLS over IPv4 tunneling driver
0.393528] NET: Registered PF_PACKET protocol family
0.394708] gmett: Installing 992800 support
0.395131] gmett: 1024 entries required
0.397087] debug_vn_pgtable: (debug_vn_pgtable) : Validating architecture pa Breakpoint 3 at 0xffffffff8016d0bc: file fs/exec.c, line 2091.
0.433099] Freeing unused kernel image (initmen) memory: 2172K
0.444948] Run /init as init process

Breakpoint 1, _se sys_execve (filename=0x458824, argv=0x72057593919702688, envp=0x72057593919702688)
2091     SYSCALL_DEFINE3(execve,
2092     const char __user *, filename,
2093     const char __user *const __user *, argv,
2094     const char __user *const __user *, envp)
2095 {
2096     return do_execve(getname(filename), argv, envp);
2097 }
2098
2099 SYSCALL_DEFINE5(execveat,
2100     int, fd, const char __user *, filename,

```

图 7.8: 调试 exec

首先跳转到 SYSCALL_DEFINE3(execve,) :

```

2085             return;
2086
2087         set_mask_bits(&mm->flags, MMF_DUMPABLE_MASK, value);
2088     }
2089
B+> 2091     SYSCALL_DEFINE3(execve,
2092     const char __user *, filename,
2093     const char __user *const __user *, argv,
2094     const char __user *const __user *, envp)
2095 {
2096     return do_execve(getname(filename), argv, envp);
2097 }
2098
2099 SYSCALL_DEFINE5(execveat,
2100     int, fd, const char __user *, filename,

```

图 7.9: SYSCALL_DEFINE3

函数体中有 do_execve()，然后使用 b do_execve 命令进行打断点：

```

2013
2014 static int do_execve(struct filename *filename,
2015     const char __user *const __user *__argv,
2016     const char __user *const __user *__envp)
2017 {
2018     struct user_arg_ptr argv = { .ptr.native = __argv };
2019     struct user_arg_ptr envp = { .ptr.native = __envp };
3+> 2020     return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
2021 }
2022
2023 static int do_execveat(int fd, struct filename *filename,
2024     const char __user *const __user *__argv,
2025     const char __user *const __user *__envp,
2026     int flags)
2027 {
2028     struct user_arg_ptr argv = { .ptr.native = __argv };

```

图 7.10: b do_execve

进入 do_execve() 后调用 do_execveat_common 函数, 现在给 do_execveat_common 函数打上断点:

```

1868 }
1869 static int do_execveat_common(int fd, struct filename *filename,
1870                               struct user_arg_ptr argv,
1871                               struct user_arg_ptr envp,
1872                               int flags)
1873 {
1874     struct linux_binprm *bprm;
1875     int retval;
1876
1877 >+ 1878     if (IS_ERR(filename))
1879         return PTR_ERR(filename);
1880
1881     /*
1882      * We move the actual failure in case of RLIMIT_NPROC excess from
1883      * setuid() to execve() because too many poorly written programs

```

图 7.11: b do_execveat_common

而大部分的工作基本上都是在 do_execveat_common 函数内完成的, do_execveat_common 函数执行完后会依次返回到 SYSCALL_DEFINE3(execve,) 中完成调用, 完成调用后 MenuOS 中的 Shell 将显示 hello world!:

```

MenuOS>>exec
exec - execve new process
This is Parent Process!
This is Child Process!
hello world!
Child Complete!
MenuOS>>exec

```

图 7.12: hello world!

但此时, 进程仍然没有结束, 下一步将进入 schedule 函数中依次执行 schedule 函数中的内容:

```

6536
6537     asmlinkage __visible void __sched schedule(void)
6538     {
6539         struct task_struct *tsk = current;
6540
6541         sched_submit_work(tsk);
6542         do {
6543             preempt_disable();
6544             __schedule(SM_NONE);
6545             sched_preempt_enable_no_resched();
6546             } while (need_resched());
6547             sched_update_worker(tsk);
6548     }
6549     EXPORT_SYMBOL(schedule);
6550
6551 */

```

图 7.13: schedule

schedule 函数结束后，将再次引发 Shell 程序引起的中断：

```
(gdb) n
Single stepping until exit from function handle_exception,
which has no line number information.
[]
```

图 7.14: Shell 程序中断

此时 MenuOS 中的 Shell 可以再次输入命令。

第8章 实验八：理解进程调度时机跟踪分析进程调度与进程切换的过程

实验要求

- 理解 Linux 系统中进程调度的时机
- 使用 gdb 跟踪分析一个 schedule() 函数
- 仔细分析 switch_to 中的汇编代码，理解进程上下文的切换机制，以及与中断上下文切换的关系；

8.1 理解 Linux 系统中进程调度的时机

进程是计算机中运行的程序的抽象，它包含了程序代码、数据、以及程序执行的上下文。进程调度的策略和算法决定哪个进程应该在某个时刻运行。进程调度的时机大多和中断有关。在多任务操作系统中，中断是一种机制，它允许操作系统在发生某些事件时打断正在执行的进程，转而执行另一个进程。这种打断和切换的机制是进程调度的基础——中断上下文恢复。中断上下文恢复是指在中断服务例程（ISR）中，由于中断事件而中断的正在执行的任务（通常是用户进程）的上下文（包括寄存器状态、程序计数器等）被保存起来，以便在中断处理完成后能够正确地恢复原来的任务继续执行。

8.2 gdb 跟踪分析 schedule() 函数

在 menuos 的目录之下，启动 init_gdb.sh 和 star-gdb.sh 两个 bash 脚本和 gdb 远程连接后，在 gdb 中输入 b schedule 命令进行对 schedule 函数进行打断点后，使用 c 命令运行到断点处。如下图所示：

```
Remote debugging using :1234
0x000000000001000 in ?? ()
(gdb) b schedule
Breakpoint 1 at 0xffffffff806a6e44: file ./arch/riscv/include/asm/current.h, line 31.
(gdb)
Note: breakpoint 1 also set at pc 0xffffffff806a6e44.
Breakpoint 2 at 0xffffffff806a6e44: file ./arch/riscv/include/asm/current.h, line 31.
(gdb) c
Continuing.

Breakpoint 1, schedule () at ./arch/riscv/include/asm/current.h:31
31          return riscv_current_is_tp;
(gdb) 
```

图 8.1: gdb schedule

在 RISC-V 架构下的 Linux 内核中，schedule 断点并没有出现在 kernel/sched/core.c 中而是出现在 arch/riscv/include/asm/current.h 中，并指向 riscv_current_is_tp。

```

./arch/riscv/include/asm/current.h
28 */
29 static __always_inline struct task_struct *get_current(void)
30 {
B+> 31     return riscv_current_is_tp;
32 }
33
34 #define current get_current()
35

0xffffffff8003119a <single_task_running+34>    add    a5,a5,a4
0xffffffff8003119c <single_task_running+36>    lw     a0,24(a5)
0xffffffff8003119e <single_task_running+38>    addi   sp,sp,16
0xffffffff800311a0 <single_task_running+40>    addi   a0,a0,-1
0xffffffff800311a2 <single_task_running+42>    seqz  a0,a0
0xffffffff800311a6 <single_task_running+46>    ret
0xffffffff800311a8 <check_same_owner>      addi   sp,sp,-16
0xffffffff800311aa <check_same_owner+2>    sd    s0,8(sp)
0xffffffff800311ac <check_same_owner+4>    addi   s0,sp,16
0xffffffff800311ae <check_same_owner+6>    lw     a4,8(tp) # 0x8

remote Thread 1.1 In: schedule                                         L31   PC: 0xffffffff806a6e44
(gdb) c
Continuing.

Breakpoint 1, schedule () at ./arch/riscv/include/asm/current.h:31
(gdb) 

```

图 8.2: riscv_current_is_tp

在进行下一步后，断点跳至 kernel/sched/core.c 中的 schedule 函数中：

```

kernel/sched/core.c
6535 }
6536
6537 asmlinkage __visible void __sched schedule(void)
6538 {
6539     struct task_struct *tsk = current;
6540
> 6541     sched_submit_work(tsk);
6542     do {

0xffffffff806a6e54 <schedule+16>    beqz  a5,0xffffffff806a6e7c <schedule+56
0xffffffff806a6e56 <schedule+18>    lw     a5,60(tp) # 0x3c
0xffffffff806a6e5a <schedule+22>    andi  a4,a5,48
0xffffffff806a6e5e <schedule+26>    bnez  a4,0xffffffff806a6ec0 <schedule+12
0xffffffff806a6e60 <schedule+28>    lw     a5,40(tp) # 0x28
0xffffffff806a6e64 <schedule+32>    slli  a4,a5,0x33
0xffffffff806a6e68 <schedule+36>    bltz  a4,0xffffffff806a6eee <schedule+17
0xffffffff806a6e6c <schedule+40>    ld    a0,1928($1)
0xffffffff806a6e70 <schedule+44>    beqz a0,0xffffffff806a6e7c <schedule+56
0xffffffff806a6e72 <schedule+46>    li    a1,1

remote Thread 1.1 In: schedule                                         L6541 PC: 0xffffffff806a6e4e
(gdb) c
Continuing.

Breakpoint 1, schedule () at ./arch/riscv/include/asm/current.h:31
(gdb) n
(gdb) 

```

图 8.3: jump to schedule

在 tsk 变量赋值后，使用 p *tsk 命令查看 tsk 的内容：

```

kernel/sched/core.c
6498         if (task_is_running(tsk))
6499             return;
6500
6501     task_flags = tsk->flags;
6502     /*
6503      * If a worker goes to sleep, notify and ask workqueue whether it
6504      * wants to wake up a task to maintain concurrency.
6505      */
6506     if (task_flags & (PF_WQ_WORKER | PF_IO_WORKER)) {
6507         if (task_flags & PF_WQ_WORKER)
6508             wq_worker_sleeping(tsk);
6509
6510     }
6511
6512     > 0xffffffff806a6e54 <schedule+16>    beqz   a5,0xffffffff806a6e7c <schedule+56
6513     0xffffffff806a6e56 <schedule+18>    lw      a5,0(0) # 0x3c
6514     0xffffffff806a6e5a <schedule+22>    andi   a4,a5,48
6515     0xffffffff806a6e5e <schedule+26>    bnez   a4,0xffffffff806a6ec0 <schedule+12
6516     0xffffffff806a6e60 <schedule+28>    lw      a5,40(0) # 0x28
6517     0xffffffff806a6e64 <schedule+32>    slli   a4,a5,0x33
6518     0xffffffff806a6e68 <schedule+36>    bltz   a4,0xffffffff806a6eee <schedule+17
6519     0xffffffff806a6e6c <schedule+40>    ld      a0,1928($1)
6520     0xffffffff806a6e70 <schedule+44>    beqz   a0,0xffffffff806a6e7c <schedule+56
6521     0xffffffff806a6e72 <schedule+46>    li      a1,1
6522     0xffffffff806a6e74 <schedule+48>    auipc  ra,0xfc2f
6523
6524
6525 remote Thread 1.1 In: schedule                                         L6541 PC: 0xffffffff806a6e54
$2 = {thread_info = {flags = 8, preempt_count = 0, kernel_sp = -2128593696,
    user_sp = -2128593696, cpu = 0}, __state = 0, stack = 0xffffffff81200000, usage = {
    refs = {counter = 2}}, flags = 69206018, ptrace = 0, on_cpu = 1, wake_entry = {
    llist = {next = 0x0}, {u_flags = 48, a_flags = {counter = 48}}, src = 0, dst = 0},
    wakee_flips = 0, wakee_flip_decay_ts = 0, last_wakee = 0x0, recent_used_cpu = 0,
    wakee_cpu = 0, on_rq = 1, prio = 120, static_prio = 120, normal_prio = 120,
    rt_priority = 0, se = {load = {weight = 1048576, inv_weight = 4194304}, run_node = {
        __rb_parent_color = 0, rb_right = 0x0, rb_left = 0x0}, group_node = {
        next = 0xffffffff8120df28 <init_task+232>,
        prev = 0xffffffff8120df28 <init_task+232>}, on_rq = 0, exec_start = 0,
        sum_exec_runtime = 0, vruntime = 0, prev_sum_exec_runtime = 0, nr_migrations = 0,
        depth = 0, parent = 0x0, cfs_rq = 0xff60000007c27300, my_q = 0x0,
        runnable_weight = 0, avg = {last_update_time = 0, load_sum = 0, runnable_sum = 0,
        util_sum = 0, period_contrib = 0, load_avg = 0, runnable_avg = 0, util_avg = 0,
        --Type <RET> for more, q to quit, c to continue without paging--}
    }
}

```

图 8.4: p *tsk

使用 b __schedule 对 __schedule 设置断点并使用 c 命令运行到断点处：

```

kernel/sched/core.c
6346 *
6347 * WARNING: must be called with preemption disabled!
6348 */
6349 static void __sched notrace __schedule(unsigned int sched_mode)
B+> 6350 {
6351     struct task_struct *prev, *next;
6352     unsigned long *switch_count;
6353     unsigned long prev_state;
6354     struct rq_flags rf;
6355     struct rq *rq;
6356     int cpu;

B+> 0xffffffff806a68b6 <__schedule>    addi    sp,sp,-112
0xffffffff806a68b8 <__schedule+2>    sd      $0,96(sp)
0xffffffff806a68ba <__schedule+4>    sd      $3,72(sp)
0xffffffff806a68bc <__schedule+6>    sd      $5,56(sp)
0xffffffff806a68be <__schedule+8>    sd      ra,104(sp)
0xffffffff806a68c0 <__schedule+10>   sd      $1,88(sp)
0xffffffff806a68c2 <__schedule+12>   sd      $2,80(sp)
0xffffffff806a68c4 <__schedule+14>   sd      $4,64(sp)
0xffffffff806a68c6 <__schedule+16>   sd      $6,48(sp)
0xffffffff806a68c8 <__schedule+18>   sd      $7,40(sp)
0xffffffff806a68ca <__schedule+20>   addi    $0,sp,112

remote Thread 1.1 In: schedule                                     L6350 PC: 0xffffffff806a68b6
_rb_parent_color = 0, rb_right = 0x0, rb_left = 0x0}, group_node = {
next = 0xffffffff8120df28 <init_task+232>,
prev = 0xffffffff8120df28 <init_task+232>, on_rq = 0, exec_start = 0,
sum_exec_runtime = 0, vruntime = 0, prev_sum_exec_runtime = 0, nr_migrations = 0,
depth = 0, parent = 0x0, cfs_rq = 0xffff60000007c27300, my_q = 0x0,
runnable_weight = 0, avg = {last_update_time = 0, load_sum = 0, runnable_sum = 0,
util_sum = 0, period_contrib = 0, load_avg = 0, runnable_avg = 0, util_avg = 0,
--Type <RET> for more, q to quit, c to continue without paging--qquit
(gdb) b __schedule
Breakpoint 2 at 0xffffffff806a68b6: file kernel/sched/core.c, line 6350.
(gdb) c
Continuing.

Breakpoint 2, __schedule (sched_mode=sched_mode@entry=0) at kernel/sched/core.c:6350
(gdb) 

```

图 8.5: b __schedule

在 __schedule 函数中使用 n 命令单独后，发现 __schedule 函数调用了 context_switch 函数：

```

kernel/sched/core.c
6464
6465         trace_sched_switch(sched_mode & SM_MASK_PREEMPT, prev, next
6466
6467         /* Also unlocks the rq: */
B+> 6468     rq = context_switch(rq, prev, next, &rf);
6469 } else {
6470     rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
6471
6472     rq_unpin_lock(rq, &rf);
6473     __balance_callbacks(rq);
6474     raw_spin_rq_unlock_irq(rq);

```

图 8.6: 调用 context_switch 函数

对 context_switch 打入断点后，单步执行后发现调用 switch_to：

```

kernel/sched/core.c
5149 }
5150
5151     rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
5152
5153     prepare_lock_switch(rq, next, rf);
5154
5155     /* Here we just switch the register state and the stack. */
> 5156     switch_to(prev, next, prev);
5157     barrier();
5158
5159     return finish_task_switch(prev);

```

图 8.7: 调用 switch_to

此时无法对 switch_to 进行打断点，因为 switch-to 是一个宏定义，在预处理阶段就已经被替换了，所以只能对 __switch_to 打断点，由于 RISC-V 架构下 __switch_to 是有纯汇编写的，所以无法展示 c 语言代码。如下图所示：

```

kernel/sched/core.c
[ No Source Available ]

B+ 0xffffffff80003520 <__switch_to>      lui    a4,0x1
      0xffffffff80003522 <switch_to+2>      addiw a4,a4,-1688
> 0xffffffff80003526 <switch_to+6>      add    a3,a0,a4
      0xffffffff8000352a <switch_to+10>     add    a4,a4,a1
      0xffffffff8000352c <switch_to+12>     sd    ra,0(a3)
      0xffffffff80003530 <switch_to+16>     sd    sp,8(a3)
      0xffffffff80003534 <switch_to+20>     sd    s0,16(a3)
      0xffffffff80003536 <switch_to+22>     sd    s1,24(a3)
      0xffffffff80003538 <switch_to+24>     sd    s2,32(a3)
      0xffffffff8000353c <switch_to+28>     sd    s3,40(a3)

remote Thread 1.1 In: switch to          L??   PC: 0xffffffff80003526
Breakpoint 3, context_switch (rf=0xff2000000060bb68, next=0xff6000001638b00, prev=0xff
0000001638b00, rq=<optimized out>) at kernel/sched/core.c:5108
(gdb) c
Continuing.

Breakpoint 4, 0xffffffff80003520 in __switch_to ()
(gdb) si
0xffffffff80003522 in __switch_to ()
(gdb) si
0xffffffff80003526 in __switch_to ()
(gdb) []

```

图 8.8: b __switch_to

8.3 分析 switch_to 中的汇编代码

在 RISC-V 架构的 Linux 内核中，switch_to 的汇编代码位于 arch/riscv/kernel/entry.S 中。

以下代码是 switch_to 部分的汇编代码：

```

ENTRY(__switch_to)
/* Save context into prev->thread */
li    a4, TASK_THREAD_RA
add  a3, a0, a4
add  a4, a1, a4
REG_S ra, TASK_THREAD_RA_RA(a3)
REG_S sp, TASK_THREAD_SP_RA(a3)
REG_S s0, TASK_THREAD_SO_RA(a3)
REG_S s1, TASK_THREAD_S1_RA(a3)

```

```

REG_S s2, TASK_THREAD_S2_RA(a3)
REG_S s3, TASK_THREAD_S3_RA(a3)
REG_S s4, TASK_THREAD_S4_RA(a3)
REG_S s5, TASK_THREAD_S5_RA(a3)
REG_S s6, TASK_THREAD_S6_RA(a3)
REG_S s7, TASK_THREAD_S7_RA(a3)
REG_S s8, TASK_THREAD_S8_RA(a3)
REG_S s9, TASK_THREAD_S9_RA(a3)
REG_S s10, TASK_THREAD_S10_RA(a3)
REG_S s11, TASK_THREAD_S11_RA(a3)
/* Restore context from next->thread */
REG_L ra, TASK_THREAD_RA_RA(a4)
REG_L sp, TASK_THREAD_SP_RA(a4)
REG_L s0, TASK_THREAD_S0_RA(a4)
REG_L s1, TASK_THREAD_S1_RA(a4)
REG_L s2, TASK_THREAD_S2_RA(a4)
REG_L s3, TASK_THREAD_S3_RA(a4)
REG_L s4, TASK_THREAD_S4_RA(a4)
REG_L s5, TASK_THREAD_S5_RA(a4)
REG_L s6, TASK_THREAD_S6_RA(a4)
REG_L s7, TASK_THREAD_S7_RA(a4)
REG_L s8, TASK_THREAD_S8_RA(a4)
REG_L s9, TASK_THREAD_S9_RA(a4)
REG_L s10, TASK_THREAD_S10_RA(a4)
REG_L s11, TASK_THREAD_S11_RA(a4)
/* The offset of thread_info in task_struct is zero. */
move tp, a1
ret
ENDPROC(__switch_to)

```

8.4 分析汇编代码

看到这段汇编代码，我们首先要做的是预处理一下，首先理解这个 REG... 和 TASH... 这些宏是什么。这里可以通过全局搜索，找到他们的定义。经过全局搜索发现，他们被定义在 arch/riscv/include/asm/include.h 里面。

```

#ifndef __ASM_RISCV_ASM_H
#define __ASM_RISCV_ASM_H

#ifdef __ASSEMBLY__
#define __ASM_STR(x) x
#else
#define __ASM_STR(x) #x
#endif

#if __riscv_xlen == 64
#define __REG_SEL(a, b) __ASM_STR(a)
#elif __riscv_xlen == 32
#define __REG_SEL(a, b) __ASM_STR(b)

```

```

#else
#error "Unexpected __riscv_xlen"
#endif

#define REG_L __REG_SEL(ld, lw)
#define REG_S __REG_SEL(sd, sw)

```

在 riscv 中一个字是 32 位，64 字是 double word。在汇编代码中一般用 w (32 位) 表示一个字，用 d (64 位) 表示 2 个字。在 c 语言的宏里面 # 的作用是将传入的字符变成一个用双引号包裹的字符串（请自行查看相关标准）。我们这里不需要双引号，并且是 64 位，所以 REG_L 预处理之后应该是 ld 和 sd

在 arch/riscv/kernel/asm-offsets.c 中我们可以找到 TASK_THREAD_RA_RA 的定义

```

void asm_offsets(void)
{
    OFFSET(TASK_THREAD_RA, task_struct, thread.ra);
    ...
    DEFINE(TASK_THREAD_RA_RA,
        offsetof(struct task_struct, thread.ra)
        - offsetof(struct task_struct, thread.ra)
    );
}

```

这个 TASK_THREAD_RA_RA 的意思就是得到 ra 寄存器距离 ra 寄存器的 offset。这里是用 task_struct 里面的 thread.ra 的地址减去 thrad.ra 的地址。而 TASH_THREAD_RA 就是 thread.ra 距离 task_struct 的偏移地址。接下来我们来看看这个 thread 的数据结构，它在/arch/riscv/include/asm/processor.h 里面。

```

struct thread_struct {
    /* Callee-saved registers */
    unsigned long ra;
    unsigned long sp; /* Kernel mode stack */
    unsigned long s[12]; /* s[0]: frame pointer */
    struct __riscv_d_ext_state fstate;
    unsigned long bad_cause;
    unsigned long vstate_ctrl;
    struct __riscv_v_ext_state vstate;
};

```

我们再来看看调用 switch_to 的代码，发现第一个参数是需要保存的 task_struct 第二个参数是需要加载的 task_struct。也就是说 a0 = prev ,a1 = next

```

#define switch_to(prev, next, last)
do {
    struct task_struct *__prev = (prev);
    struct task_struct *__next = (next);
    if (has_fpu())
        __switch_to_aux(__prev, __next);
    ((last) = __switch_to(__prev, __next));
} while (0)

```

让我们以伪代码的形式来理解这段代码。

```
a0 = prev
```

```

a1 = next
offset = &task_struct - &(task_struct->ra)//ra距离task_struct偏移
ENTRY(__switch_to)
/* Save context into prev->thread */
li    a4, offset
add  a3, a0, a4 //a3 = prev->thread.ra
add  a4, a1, a4 //a4 = next->thread.ra
sd ra, a3
sd sp, 8(a3)
sd s0, 16(a3)
sd s1, 24(a3)
sd s2, 32(a3)
sd s3, 40(a3)
sd s4, 48(a3)
sd s5, 56(a3)
sd s6, 64(a3)
sd s7, 72(a3)
sd s8, 80(a3)
sd s9, 88(a3)
sd s10, 96(a3)
sd s11, 104(a3)
/* Restore context from next->thread */
ld ra, 0(a4)
ld sp, 8(a4)
ld s0, 16(a4)
ld s1, 24(a4)
ld s2, 32(a4)
ld s3, 40(a4)
ld s4, 48(a4)
ld s5, 56(a4)
ld s6, 64(a4)
ld s7, 72(a4)
ld s8, 80(a4)
ld s9, 88(a4)
ld s10, 96(a4)
ld s11, 108(a4)
/* The offset of thread_info in task_struct is zero. */
move tp, a1 // 将线程寄存里面呢的值改为当前的task_struct的地址
ret
ENDPROC(__switch_to)

```

转化为这种代码之后，我们就非常容易理解了，只要会读 riscv 的汇编（请参阅 lab2）就能理解代码的意思。这里简单解释一下，就是将当前线程的 ra, sp, s0-s11 存入 prev->thread 部分。然后将 next->thread 里面的值加载进寄存器，最后切换线程指针 tp。

为什么只需保存这么少？因为我们只需要保存 caller save 的寄存器就够了，其他的寄存器是不影响的。如果需要详细理解，请参阅 lab2 里面的链接

8.5 理解进程上下文的切换机制和中断上下文切换的关系

进程上下文切换是指挂起现在正在运行的进程，并恢复之前挂起的某个进程。恢复一个之前挂起的进程必须使得寄存器保存现在被挂起的进程的值，而这部分需要保存的值被称为进程上下文。

进程执行的环境切换分为两步：

1. 从就绪队列中选择一个进程
2. 完成进程的上下文切换

这两个步骤分别涉及 `pick_next_task` 和 `context_switch` 两个函数。

中断是一种异步事件，可以打断当前正在执行的任务，跳转到中断服务例程（ISR）来处理中断。在 RISC-V 架构中，中断通常被称为“异常”（exceptions）。RISC-V 将中断、陷阱（traps）、系统调用（system calls）等都统一称为异常。

RISC-V 的异常处理机制包括三种类型的异常：

1. 中断（Interrupts）：由外部事件引起，例如时钟中断、I/O 中断等。
2. 陷阱（Traps）：由程序内部事件引起，例如软件陷阱，通常用于实现系统调用。
3. 故障（Faults）：由于非法操作或错误引起的异常，例如页错误。

中断上下文切换是指在处理中断时，保存当前任务的上下文，执行中断服务例程，然后在处理完成后，恢复原任务的上下文。进程上下文的切换机制和中断上下文切换的相似之处：

1. 都需要停止当前进程并保存当前停止的任务的上下文
2. 保存当前任务的上下文后，都需要切换上下文
3. 保存的上下文，在将来的某个时段都会重新恢复运行

而不同点在于：

1. 触发时机不同：进程上下文切换通常是由操作系统的调度器在特定时机决定的，例如时钟中断、I/O 中断等。而中断上下文切换是由硬件中断信号触发的，例如定时器中断、硬件故障中断等。
2. 执行过程不同：进程上下文切换包括保存和加载两个阶段，而中断上下文切换主要涉及中断服务例程的执行过程。
3. 保存和恢复的内容不同：进程上下文切换需要保存和恢复整个进程的执行环境，包括寄存器、内存页表等。中断上下文切换主要关注于保存和恢复与中断服务例程执行相关的上下文。

后记

千呼万唤始出来。首先祝贺读者顺利完成八个实验的复现。在你的努力和付出中，我们看到了坚持不懈的精神和对科学事业的执着。这八次实验的成功不仅仅是本手册的胜利，更是你在科研道路上不断前行的证明。你的毅力和智慧在这一过程中得到了充分展现，我们对此感到自豪。这次复现实验的成功，将为我们未来的工作提供坚实的基础。再次恭喜你完成这八次实验，希望你的科研之路越来越宽广，未来的实验取得更加令人瞩目的成果。期待在共同的科研征途中，我们能够一同奋斗，创造更多的科学奇迹。

本手册使用 \LaTeX 进行编写，主题使用了 [ElegantBook Template](#)。感谢 \LaTeX 社区和 ElegantBook 的作者们。本手册能够完成，是因为有我的合伙人 ZyLqb 的共同努力，有 605 的师兄师弟们的热心帮助，有我的导师张健毅教授的支持，有帮我们办材料和募集资金的同学的积极帮助，更是因为娄嘉鹏老师在设备和资金上的大力支持。在整个过程中，ZyLqb 作为我的合伙人，我们携手合作，共同克服了种种困难，让这个手册得以顺利完成。再次感谢娄嘉鹏老师，在设备和资金方面的慷慨支持让我们能够顺利进行实验和研究，为手册提供了坚实的资金和设置基础。在科研道路上，有了这样的支持和帮助，我们更加有信心和动力前行。

感谢合作伙伴的支持，也感谢娄家鹏老师的大力资助。在未来的研究和工作中，我们将一如既往地努力拼搏，为科学事业贡献我们的力量。