Lab1 最小可执行内核

实验目的:

实验 1 主要关于最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行,它可以模拟一台 64 位 RISC-V 计算机。为了让内核能够正确对接到 Qemu 模拟器上,需要了解 Qemu 模拟器的启动流程,还需要一些程序内存布局和编译流程(特别是链接)相关知识。

本章将学习:

- 使用链接脚本描述内存布局
- 进行交叉编译生成可执行文件,进而生成内核镜像
- 使用 OpenSBI 作为 bootloader 加载内核镜像,并使用 Qemu 进行模拟
- 使用 OpenSBI 提供的服务,在屏幕上格式化打印字符串用于以后调试

练习 1: 理解内核启动中的程序入口操作

阅读 kern/init/entry.S 内容代码,结合操作系统内核启动流程,说明指令 la sp, bootstacktop 完成了什么操作,目的是什么? tail kern_init 完成了什么操作,目的是什么?

kern/init/entry.S 内容代码如下:

```
.section .text, "ax", %progbits
```

.globl kern entry

kern_entry:

la sp, bootstacktop # 设置栈指针 SP 为 bootstacktop tail kern_init # 跳转到 C 语言初始化函数 kern_init

.section .data

.align PGSHIFT

.global bootstack

bootstack:

.space KSTACKSIZE

分配内核栈空间

.global bootstacktop

bootstacktop: # 栈顶地址

1.la sp, bootstacktop

la 是"load address"的伪指令,它会把标签 bootstacktop 对应的内存地址加载到寄存器 sp(stack pointer)中,相当于 sp = &bootstacktop,其作用是为内核设置一个栈顶地址,分配启动时所需的运行时栈空间。因为 C 语言函数调用依赖栈保存返回地址和局部变量,所以在跳转到任何 C 代码之前必须先初始化栈。栈指针 sp 被设置为内核启动栈(bookstack)的顶部地址,初始化启动栈,使得之后 C 语言内核初始化函数 kern_init 执行时,函数调用、局部变量等都能正常使用栈空间。

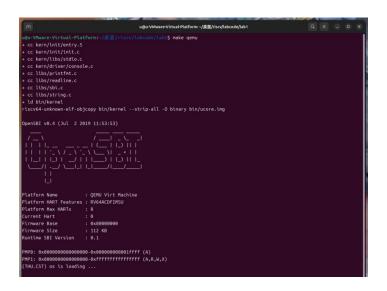
2.tail kern_init

tail 也是一个 RISC-V 的伪指令,它表示"无返回地跳转到另一个函数",所以 tailkern_init 相当于无返回调用 kern_init(类似 jmp)内核初始化函数,使 得从汇编启动代码正式跳转到 C 语言内核入口并且不返回启动代码。tail 指令 会释放当前函数的栈帧,相当于优化版的 j kern_init,不会再返回到 kern entry。由这一步进入 C 语言部分,内核从汇编正式转交给 C 初始化函数。

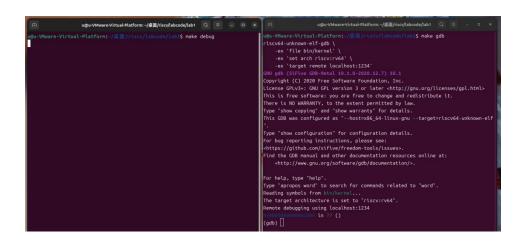
练习 2: 使用 GDB 验证启动流程

为了熟悉使用 QEMU 和 GDB 的调试方法,请使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始,直到执行内核第一条指令(跳转到 0x80200000)的整个过程。通过调试,请思考并回答: RISC-V 硬件加电后最初执行的几条指令位于什么地址?它们主要完成了哪些功能?请在报告中简要记录你的调试过程、观察结果和问题的答案。

1. 首先在终端中运行 make qemu 命令,启动 QEMU,编译并生成了内核镜像文件 bin/ucore.img:



2. 然后开始调试,在lab1文件夹中打开两个终端窗口。左边窗格我们输入make debug,这个命令会启动QEMU;右边窗格,我们输入make gdb,启用gdb 调试:



在 GDB 调试器中,使用 x/10i \$pc 命令查看 PC 寄存器指向的指令,显示了从地址 0x1000 开始指令的汇编代码:

```
(gdb) x/10i $pc
               auipc
                       t0,0x0
               addi
                       a1,t0,32
               CSFF
                       a0, mhartid
               ld
                       t0,24(t0)
                       t0
               unimp
               unimp
               unimp
               0x8000
               unimp
(gdb)
```

发现在 0x1010 处指令跳转到地址 0x80000000, 于是使用指令 si 单步执行, 每步后使用指令 i r t0 (info registers t0) 查看寄存器结果:

```
00000001004 in ?? ()
(gdb) i r t0
t0
              0x1000 4096
(gdb) si
(gdb) i r t0
t0
              0x1000 4096
(gdb) si
    0000000000100c in ?? ()
(gdb) i r t0
              0x1000 4096
t0
(gdb) si
     0000000001010 in ?? ()
(gdb) i r t0
              0×80000000
                               2147483648
t0
(gdb) si
(gdb)
```

发现在执行到地址 0x1010 时, t0=[t0+24], 寄存器被赋予了新值。

于是地址跳转到 0x80000000, 使用命令 x/10i 0x80000000 查看指令的汇编代码:

```
(gdb) x/10i 0x80000000
                      a6,mhartid
                      a6,0x80000108
   0x80000004: bgtz
  0x80000008: auipc t0,0x0
   0x80000000c: addi
                      t0,t0,1032
  0x80000010: auipc
                      t1,0x0
                      t1,t1,-16
     80000014: addi
                      t1,0(t0)
  0x80000018: sd
  0x8000001c: auipc
                      t0,0x0
  0x80000020: addi
                      t0,t0,1020
              ld
                      t0,0(t0)
(dbp)
```

- 3. 发现内核代码开始执行。链接脚本指定了内核的入口地址,我们可以直接在这个地址上打断点,但更简单的方法是使用函数名。所以使用命令 b* kern_entry 对 kern_entry 函数设置断点:
- 4. 之后使用命令 c 执行程序,内核在运行到我们设置好的断点处停止。 kern_entry 的作用为分配好内核栈,此时内核暂停在入口函数的第一条汇 编指令处:

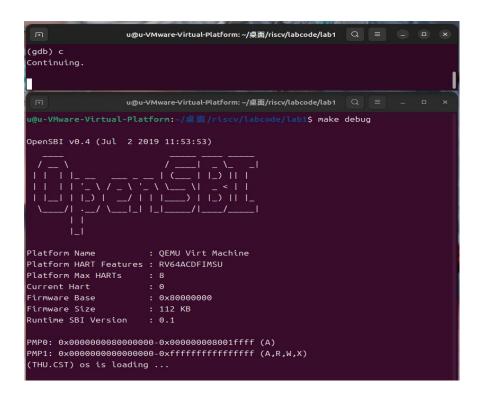
此时可以使用命令 ir 查看所有寄存器的值:

```
(gdb) i r
га
                0x80000a02
sp
                0x8001bd80
gp
                0x0
tp
                0x8001be00
t0
                0x80200000
                                  2149580800
t1
                0x1
t2
                0x1
                0x8001bd90
```

然后会跳转到 kern_init, kern_init 函数会初始化内核环境,于是使用命令 b* kern init 在此处打断点,之后也可以使用命令 si 逐步观察:

```
(gdb) b* kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7 la sp, bootstacktop
(gdb) si
9 tail kern_init
```

- 5. 然后使用命令 disassemble kern_init 查看反汇编代码,发现最后一个指令j 0x8020003c <kern_init+48>会跳转到自己,从而实现循环。
- 6. 再使用命令 c 执行程序, 出现 os is loading...:



若不构建内核镜像

1. 启动 QEMU 并等待 GDB 连接

指令: qemu-system-riscv64 -machine virt -nographic -bios default -kernel bin/kernel -S -s

2. 在另一个终端启动 GDB, bin/kernel 是编译的内核文件

指令: riscv64-unknown-elf-gdb bin/kernel

根据环境配置的不同也可能是: gdb-multiarch bin/kernel

3. 连接到 QEMU 后开始调试

gdb 连接上 QEMU 的 RISC-V 虚拟 CPU 调试接口,如下图

CPU 从复位地址(0x1000)开始执行初始化固件(OpenSBI)的汇编代码,此时 QEMU 虚拟 CPU 还没执行任何固件,处于刚加电的状态。

● 输入 info registers 可以查看寄存器的初始状态,记录了一些寄存器的初始值:

寄存器	初始值	含义
sp	0x0	尚未建立栈空间
рс	0x1000	RISC-V 的复位地址
ra	0x0	未初始化,当前没有调用栈环境

● 设置断点在内核入口 b *0x80200000

0x80200000 是内核镜像的加载地址,也就是 kern_entry 所在的位置。

● 反汇编出从 0x1000 开始的 10 条指令,输入指令 x/10i \$pc:

```
(gdb) x/10i $pc
=> 0x1000:
                auipc
                         t0,0x0
                addi
                         a1,t0,32
                CSTT
                         a0, mhartid
                0x182b283
  0x100c:
                jг
                unimp
                unimp
  0x1018:
                unimp
  0x101a:
                0x8000
                unimp
```

CPU 从复位向量 0x1000 开始执行。0x1000 处的一小段跳板代码首先用 auipc / addi 构造固件基址、设置跳转目标,然后用 csrr a0, mhartid 读取 hart id 并把参数放到寄存器(如 a1),最后通过 jr t0 将控制权跳转到 OpenSBI 入口地址。因此,加电后最早执行的几条指令的主要功能是建立跳转目标并将控制权交给固件,而那些在反汇编中显示为 unimp 或原始字的部分通常是用于构造地址的立即数字段或数据/对齐填充(可能因 GDB 的解码/架构设置而未被识别)。

● 通过设置断点,来查看 t0 的值,验证跳转目标:

```
(gdb) b *0x1010

Breakpoint 1 at 0x1010
(gdb) c

Continuing.

Breakpoint 1, 0x00000000001010 in ?? ()
(gdb) info registers t0
t0 0x80000000 2147483648
```

● 继续执行直至跳转到内核:

OpenSBI 已完成初始化,控制权转移到了内核。

在 GDB 中查看 0x80200000 内存前 64 字节时,发现全是 0x00,说明当前 QEMU 实例中内核还未被加载。

这是因为启动时没有指定内核镜像(-kernel 参数),因此 PC 指向内核入口处时,GDB 无法识别指令,反汇编显示 unimp 。

只有指定内核镜像并加载到内存后,才能在 GDB 中正确看到内核入口指令。

重要知识点

- 1. 内核启动流程(Boot)
- OS 原理对应:操作系统启动(Booting)
- 含义/关系:实验中通过 OpenSBI + entry.S 演示 RISC-V 内核启动, OS 原理中启动流程涉及从 BIOS/固件到内核加载,但实验简化了硬件的复杂性,只模拟一台 CPU。
- 2. 栈初始化(la sp, bootstacktop)
- OS 原理对应:函数调用和栈机制
- 含义/关系:内核启动时必须初始化栈来支持C函数调用。栈用于保存返回地址、局部变量和参数,本次实验中用固定大小启动栈,简化了多线程/进程栈的概念。
- 3. 无返回调用 (tail kern init)
- OS 原理对应:函数调用/返回机制
- 含义/关系: tail 跳转类似 0S 原理中的函数调用优化,不返回调用起始汇编代码,保证内核从汇编进入 C 入口顺利。
- 4. 链接脚本 (kernel.ld)
- OS 原理对应:内存管理与程序加载
- 含义/关系:链接脚本决定各段在内存的布局(text, data, bss),加载器负责地址映射、段管理。本次实验简化,没有动态分区/分页,只固定内存地址。
- 5. OpenSBI
- OS 原理对应:系统初始化
- 含义/关系: OpenSBI 提供基础硬件初始化和服务,类似 Bootloader 的概念。差异是本次实验只处理单 CPU、单核、固定外设,没有复杂中断和外设管理。
- 6. PC 寄存器
- 含义:程序计数器,记录当前执行指令地址,控制 CPU 执行流,PC 决定指令顺序执行。实验中通过 GDB 查看 PC 值,理解内核从固件跳转到内核入口的过程。

0S 原理中重要但实验未涉及的知识点

● 多核/多线程启动

内核需要启动多个处理器核,协调任务,本次实验只模拟了单核,简化启动流程

● 中断和异常处理

即硬件事件响应机制,但本次实验中内核还未注册异常和中断处理函数,只是最小可执行内核

● 动态内存管理(分页/虚拟内存)

现代 OS 使用虚拟内存隔离进程,实验内核静态分配启动栈和数据段,没有 MMU 配置