实验报告：最小可执行内核与启动流程

一、 实验目的

实验1主要讲解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。为了让我们的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。

本章你将学到：

1. 使用 链接脚本 描述内存布局
2. 进行 交叉编译 生成可执行文件，进而生成内核镜像
3. 使用 OpenSBI 作为 bootloader 加载内核镜像，并使用 Qemu 进行模拟
4. 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

二、 实验环境

- **操作系统**：Ubuntu 24.04
- **模拟器**：QEMU 4.1.1 for RISC-V 64
- **固件**：OpenSBI v0.4
- **工具链**：riscv64-unknown-elf-gcc交叉编译工具链

三、 实验内容与结果

1. 最小内核编译与运行

通过`make qemu`指令编译内核并启动QEMU，成功观察到OpenSBI加载并运行，证明实验环境配置正确，内核可以被固件识别。

```
samar1@LAPTOP-V1NFR0ES:~/labcode/lab1$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

2. 重要知识点梳理

2.1 实验知识点与OS原理对应关系

实验知识点	OS原理知识点	实验含义	原理含义	关系与差异
内核入口点 (`kern_entry`)	**系统启动流程 (Bootstrapping)**	内核代码的起始执行点，在本项目中是`0x80200000`处的汇编代码。	操作系统代码在被引导程序加载后，开始执行的第一条指令。	**关系**：实验中的`kern_entry`是OS启动流程中，控制权从Bootloader（OpenSBI）转移到OS内核的具体承接点。 **差异**：原理描述的是一个抽象的、多阶段的过程（BIOS->MBR->Bootloader->Kernel），而实验则具体展示了从固件跳转到内核入口这一关键环节的实现细节。
内存布局 (链接脚本)	**内存管理 (Memory Management)**	通过链接脚本（`ld.script`）静态规定内核的`.text`，`.data`，`.bss`等段在物理内存中的确切位置。	操作系统如何组织和管理内存空间，包括物理内存和虚拟内存。	**关系**：链接脚本是实现内核静态内存布局的工具，是内存管理的最早阶段。 **差异**：本实验只涉及内核自身的**静态物理内存布局**。而OS原理中的内存管理是一个宏大且动态的概念，还包括了虚拟内存、分页、分段、动态内存分配（堆）等复杂机制。

实验知识点	OS原理知识点	实验含义	原理含义	关系与差异
栈指针初始化	**进程/线程上下文与栈管理**	在`entry.s`中，为内核的第一个执行流（C代码`kern_init`）手动设置一个有效的栈顶地址。	操作系统为每一个独立的执行单元（进程/线程）创建并管理其私有的运行时栈，用于函数调用和局部变量存储。	**关系**：实验中的初始化是为“0号进程”（即内核主线程）创建初始上下文的关键一步。 **差异**：实验是一次性的静态设置，服务于整个内核。原理中的栈管理是动态的，OS需要在创建每个新进程/线程时为其分配和设置栈。
固件交互 (OpenSBI)	**系统引导 (Bootloader) / 系统调用**	内核运行在S-Mode，通过`ecall`指令请求M-Mode的OpenSBI固件提供底层服务（如打印字符）。	计算机启动时运行的引导程序（如BIOS/UEFI, GRUB），负责硬件初始化和加载OS。广义上也包括OS与固件间的接口。	**关系**：OpenSBI在本实验中扮演了**引导程序**和**底层硬件抽象层**的双重角色。 **差异**：传统的PC启动链更长，分工更明确。RISC-V的SBI规范使得OS内核（S-Mode）和底层固件（M-Mode）的界限非常清晰，交互方式类似一种特殊的“系统调用”。

2.2 OS原理中重要但在本实验中未体现的知识点

- **虚拟内存**：整个实验过程内核都工作在物理地址上，尚未建立页表和开启MMU。
- **进程与调度**：内核是单一的执行流，没有多进程/线程的概念，因此不存在调度问题。
- **中断处理**：虽然启动流程涉及中断的使能和屏蔽，但没有建立完整的中断处理程序来响应外部设备事件。
- **并发与同步**：由于是单执行流，不存在资源竞争，因此不需要锁、信号量等同步机制。
- **文件系统**：内核镜像是作为二进制文件直接被加载到内存，没有从磁盘等存储设备读取文件的过程。

四、 练习解答

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`中的代码如下：

```
.section .text, "ax", %progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop
    tail kern_init

.section .data
.align PGSHIFT
.global bootstack
bootstack:
    .space KSTACKSIZE
.global bootstacktop
bootstacktop:
```

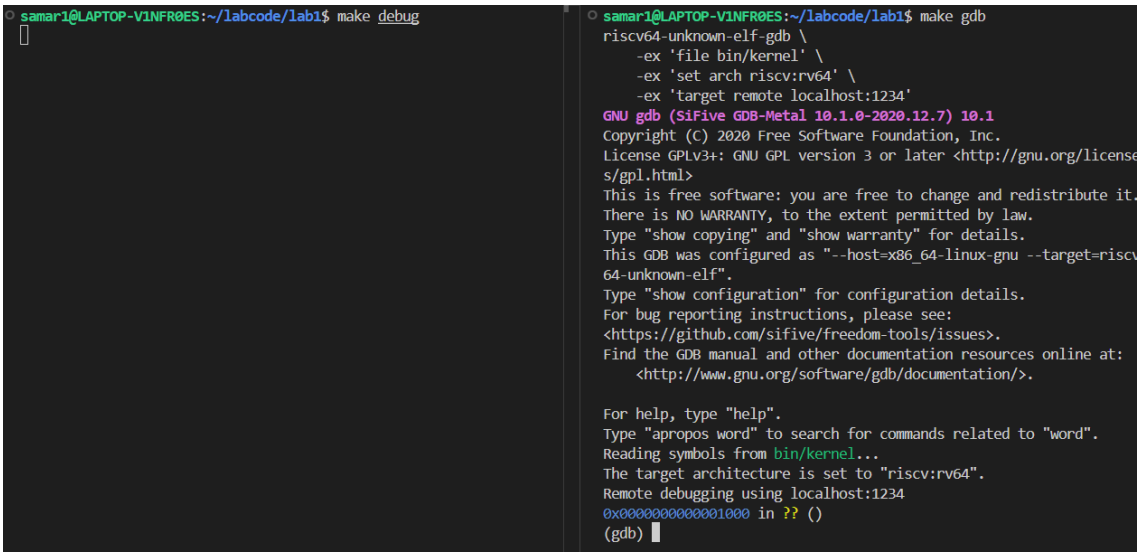
1. **la sp, bootstacktop 完成了什么操作，目的是什么？**
    - **操作**：la sp, bootstacktop是一条RISC-V伪指令，其全称为"Load Address"。它将符号bootstacktop的**内存地址**加载到栈指针寄存器sp中。bootstacktop是链接脚本中定义的bootstack栈空间的顶部（最高地址）。
    - **目的**：**为内核的C语言执行环境初始化栈。**C语言的函数调用、参数传递和局部变量存储都依赖于栈。在执行任何C函数（如此处的kern_init）之前，必须将sp指向一个有效的、可写的内存区域。这一步是连接汇编启动代码和C语言主体代码的桥梁。
2. **tail kern_init 完成了什么操作，目的是什么？**
    - **操作**：tail kern_init是一条尾调用伪指令，它会被汇编器翻译成一条无条件跳转指令 j kern_init。它会直接将PC（程序计数器）设置为kern_init函数的地址，但与常规的call（或jal）指令不同，它**不会保存返回地址**到ra寄存器。
    - **目的**：**将控制权从汇编入口转移到内核的C语言主函数kern_init。**因为entry.S中的初始化任务已经完成，程序流程不会再返回到这里，所以保存返回地址是多余的。使用尾调用（跳转）是一种简洁高效的方式，它避免了在栈上创建一个无用的栈帧，从而节省了栈空间和指令周期。

练习2: 使用GDB验证启动流程

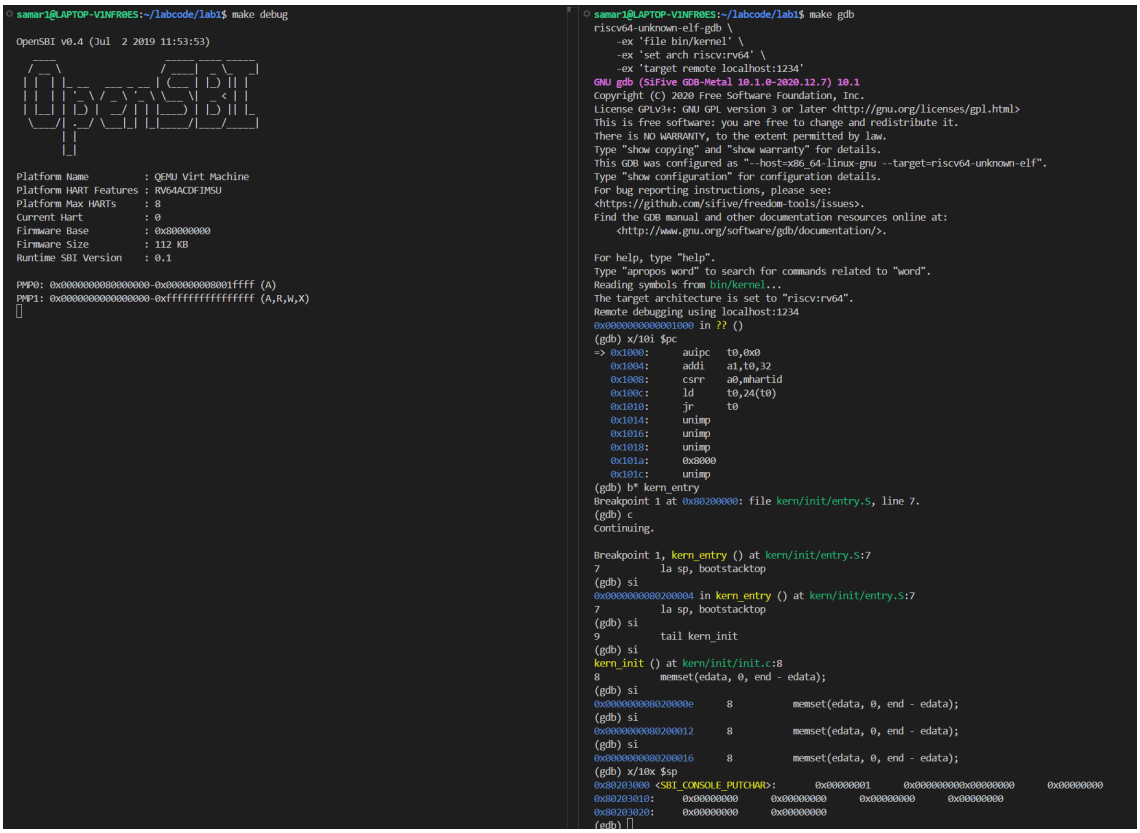
为了熟悉使用 QEMU 和 GDB 的调试方法，请使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始，直到执行内核第一条指令（跳转到 0x80200000）的整个过程。通过调试，请思考并回答：RISC-V 硬件加电后最初执行的几条指令位于什么地址？它们主要完成了哪些功能？请在报告中简要记录你的调试过程、观察结果和问题的答案。

实验过程：

1. **启动调试会话**：在一个终端窗口中运行 make debug，QEMU启动并暂停，等待GDB连接。在另一个窗口中运行 make gdb，启动GDB客户端并自动连接到QEMU。



2. **探索起点**：连接成功后，GDB默认停在程序的第一条指令。使用 x/10i $pc 命令查看当前PC（0x1000）开始的10条指令。



3. **跳转到内核**：在GDB中，我们在内核入口地址kern_entry（即0x80200000）处设置一个断点：b *0x80200000。然后使用 c (continue) 命令让程序继续执行。程序会执行完OpenSBI的初始化和加载内核部分，最终在我们的断点处停下。

4. **单步跟踪入口代码**：在断点处，我们已经进入了kern/init/entry.S。使用 si (step instruction) 命令单步执行汇编指令。可以观察到la sp, bootstacktop指令执行后，sp寄存器的值被设置为bootstacktop的地址。接着执行tail kern_init，程序会跳转到kern/init/init.c文件中的kern_init函数。

- **RISC-V硬件加电后最初执行的几条指令位于什么地址?**

  通过GDB调试观察到，QEMU模拟的RISC-V virt机器加电后，PC的初始值为 **0x1000**。因此，最初执行的指令位于该地址。这是由硬件（或模拟器）定义的复位向量（Reset Vector）。

- **它们主要完成了哪些功能?**

  从0x1000开始的指令序列是**OpenSBI固件的入口点**，它们是整个启动链的最前端，主要完成了以下底层准备工作：

地址	指令	功能分析
0x1000	auipc t0, 0x0	将PC+0的值（即0x1000）加载到t0，用于PC相对寻址。
0x1004	addi a1, t0, 32	计算一个目标地址（0x1000 + 32 = 0x1020），可能指向设备树或其他重要数据。
0x1008	csrr a0, mhartid	读取当前CPU核心的ID到a0寄存器，多核系统中用于区分不同核心。
0x100c	ld t0, 24(t0)	从t0 + 24（即0x1018）地址处加载一个8字节的值到t0。
0x1010	jr t0	跳转到t0寄存器中的地址去执行。这是一个间接跳转。

总结：这一小段汇编代码的核心功能是：**进行最基本的CPU状态设置（如获取核心ID），然后计算并跳转到OpenSBI固件中更复杂的C语言初始化部分。**它完成了从固定的硬件复位地址到固件主逻辑的“接力”过程。后续的OpenSBI代码才会真正去初始化内存、加载我们的内核镜像到0x80200000，并最终跳转过来。