LAB1:最小可执行内核ucore

实验环境

名称	版本信息
qemu	4.1.1
gcc	gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
Virtual Machine	WSL2
OS	Ubuntu 11.4.0-1ubuntu1~22.04.2
Editor	Vscode

练习1:理解内核启动的程序入口

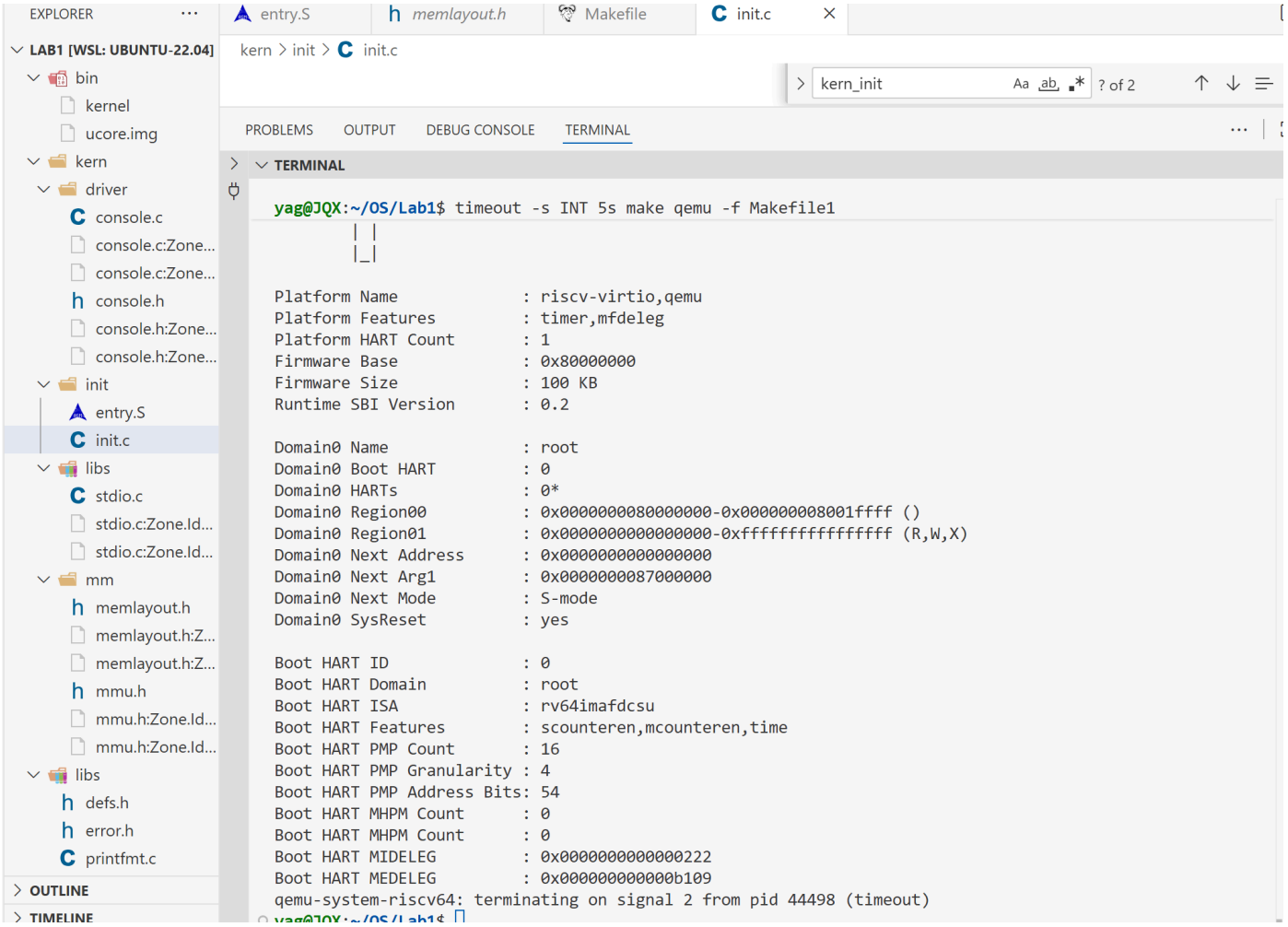
运行ucore

先尝试运行一遍完整的ucore ,观察输出, 对ucore的加载建立初步的认识.

但是,遇到了问题:

遇到的问题

用指令*make　qemu*运行, 对源码进行编译的.o文件和二进制可执行文件ELF



运行命令 `make qmeu` 没有得到os is loading ,故猜测是上一次运行被中断,残留的中间文件导致再次运行失败. 因此用命令 `make dist-clean`删除中间文件,再次运行.

然而,还是运行失败.

仔细观察会发现,`domian0 Next Address` 的值为0x00000000 ,与预期的0x80200000不符合

openSBI 的下一跳地址没有按照预期被设定为0x80200000.

找到Makefile文件中指定内核位置的代码:

```
.PHONY: qemu
```

```
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#       $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
        $(V)$(QEMU) \
                -machine virt \
                -nographic \
                -bios default \
                -device loader,file=$(UCOREIMG),addr=0x80200000
```

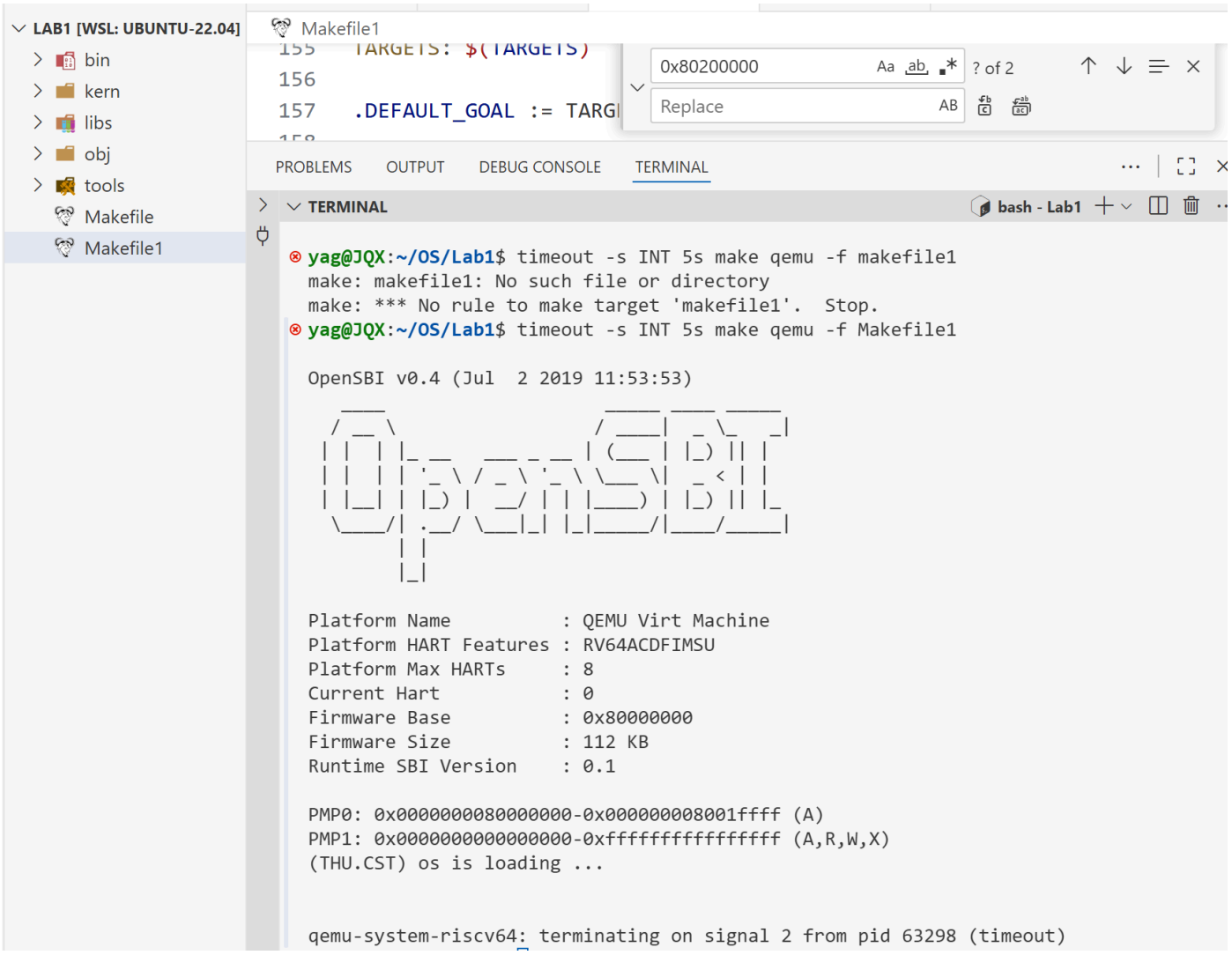
在我当时的 QEMU 版本（6.2.0）上，`-device loader` 中 `addr` 指定的地址只是将内核加载到物理地址 0x80200000，但此时处于 M 模式运行的固件 OpenSBI 并不知道下一跳的位置，也没有明确的程序控制权转移地址，因此程序停滞。

解决方案

1. 将qemu版本更换到4.1.1
2. 更改上述提到的加载内核的指令. 具体来说是:将 `-device loader,file=$(UCOREIMG),addr=0x80200000` 替换成 `kernel(kernel)` .

> 方法二不需要改变qemu的版本号,依然采用6.2.0版本
>
> `kernel(kernel)`指令作用是告诉open SBI将要加载的内核是二进制文件kernel , 在这个过程中,open SBI自动的将mecp寄存器的值置为0x80200000, 当openSBI将程序控制权转移给内核时, mecp将值赋值给pc,实现程序的跳转

运行结果



> 此处为了解决init.c中的无限循环导致终端卡死, 采用指令 `timeout -s INT 5s make qemu` ,在程序运行超过5s后杀死进程

练习问题回答

问题一回答:

1. **完成的操作**:`la sp, bootstacktop` 的作用是：把符号 bootstacktop 对应的内核栈顶地址加载到 sp 寄存器中， 从而完成内核栈指针的初始化。这为后续进入 C 代码执行函数调用提供了栈空间支持。 `la` 是伪指令 (load address)，它加载的是“地址常量”，而不是去读取内存数据。
2. **目的** :因为在内核进入 C 语言代码之前，必须有一块栈空间，否则函数调用、局部变量都无法工作.就无法使c语言编写的内核成功运行起来.

问题二回答:

1. `tail kern_init`完成的操作: tail本身是一条伪指令,与 `jal kern_init`等价,即 **无条件跳转到** `kern_init` ，并且不会返回到原来的位置。与 `jal` 不同，`tail` 并不

会保存返回地址到 `ra`，因为这是一次 **不可返回的控制权移交** 。完成了控制权的转移,将控制权转移到c语言构建的ucore内核.

> 这里有一点需要说明: kern_init的定义不在当前entry.s的汇编文件, 而是在同级目录的init.c文件中. 在entry.s中并没有显示的导入kern_init这个函数的声明,这点是有区别与c语言的习惯的.原因是因为c语言使用外部的全局函数或者变量,需要用extern声明才能通过编译阶段得到对应的汇编代码,而对于汇编代码本身使用外部全局函数,是在链接阶段链接.o文件实现. 对于 `tail kern_inti`中的kern_init,在链接阶段从init.c汇编得到的.o文件可以找到实现.

2. `tail kern_init`的目的:从汇编入口无缝过渡到C语言内核初始化代码，而不浪费栈空间（无调用帧）.kern_init负责核心初始化（如kprintf输出"kernel is booting"、内存布局计算、trap向量设置），最终调用main进入OS主循环.没有返回设计是因为这是单向引导（内核无“上层”返回）.这优化了早期代码，符合OS引导的线性流程.

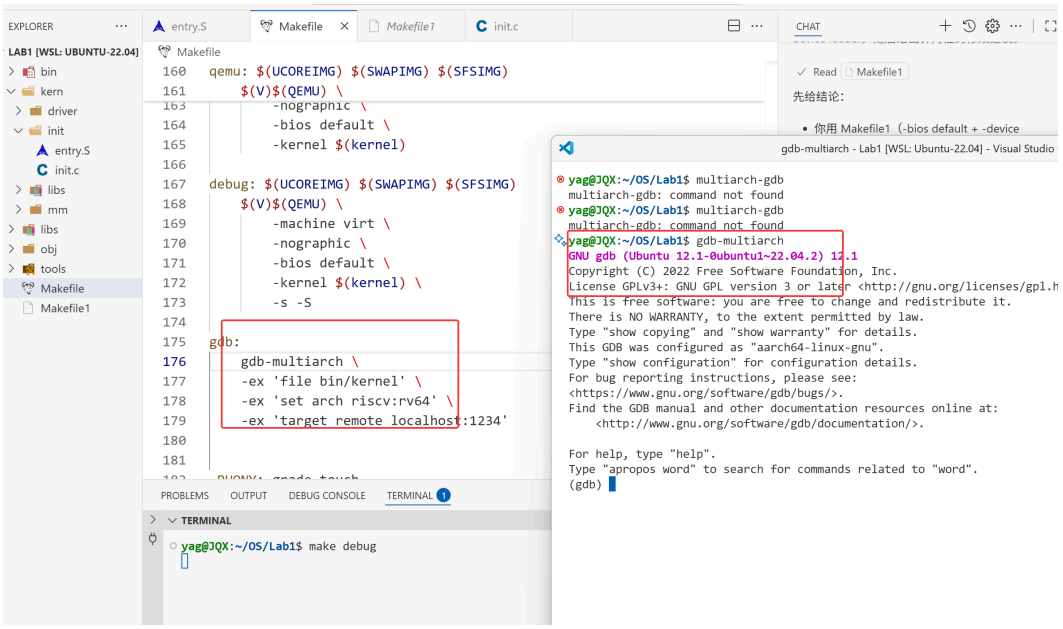
练习2:GDB验证自动启动流程

GDB进行debug

在makefile文件中已经写好了debug的自动化脚本,用 `make debug`即可调用.对应的makefile中的代码为:

```
debug: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
        $(V)$(QEMU) \
                -machine virt \
                -nographic \
                -bios default \
                -kernel $(kernel) \
                -s -S
```

- -S是CPU加电后保持暂停,不执行指令,等待我连接GDB
- -s 等价 `-gdb top::1234`,在TCP的1234端口打开GDB的服务器

gdb调试的问题	说明
	由于gdb-riscv64-unknown-elfw无法安装, 尝试了apt安装, 源码tar 下载本地编译, 以及下载已编译好 二进制文件均安装失败,故采用gdb-multiarch代替, 并修改makefile文件

之后采用 `gdb-multiarch`进行调试. 首先在一个终端里面运行 `make debug`,启动qemu进行模拟,并开放gdb服务器的默认端口1234. 之后在另外一个终端,运行 `gdb-multiarch`进行gdb调试.

在gdb的命令界面依次输入:

```
bash
file bin/kernel
set arch riscv:rv64
target remote localhost:1234
```

运行上述代码后能够看到,gdb调试窗口输出:`0x0000000000001000 in ?? ()`.

> QEMU 的引导流程和真实 RISC-V 芯片在逻辑上是一致的（CPU 从复位地址进入固件，再跳转到内核），但具体复位地址位置由硬件厂商决定，QEMU 为了简化实验统一采用 `0x1000`。ROM 作为物理地址空间的一部分，必须占据实际的地址范围，因此复位地址（如 `0x1000`）本质上就是 ROM 在物理内存中的映射位置。

开始调试:

用 `x/10i $pc`查看复位后接下来10条指令:

复位后前10条指令	说明
Breakpoint 2 at 0x80200000: file kern/init/entry.S, line 7. (gdb) x/10i $pc => 0x1000: auipc t0,0x0 0x1004: addi a1,t0,32 0x1008: csrr a0,mhartid 0x100c: ld t0,24(t0) 0x1010: jr t0 0x1014: unimp 0x1016: unimp 0x1018: unimp 0x101a: .2byte 0x8000 0x101c: unimp (gdb) info registers # 查看当前寄存器状态 x/10i $pc # 查看PC附近的汇编	计算 偏移地址加载hart ID,并跳转到固件,执行固件代码.

此时用 `info registers`查看当前的寄存器值,发现32个寄存器,只有pc有值,其余均为0.pc的值为0x00001000

> 普通寄存器不同，PC 在复位时不会是 0，而是被硬件 **强制装载为复位向量地址** 。

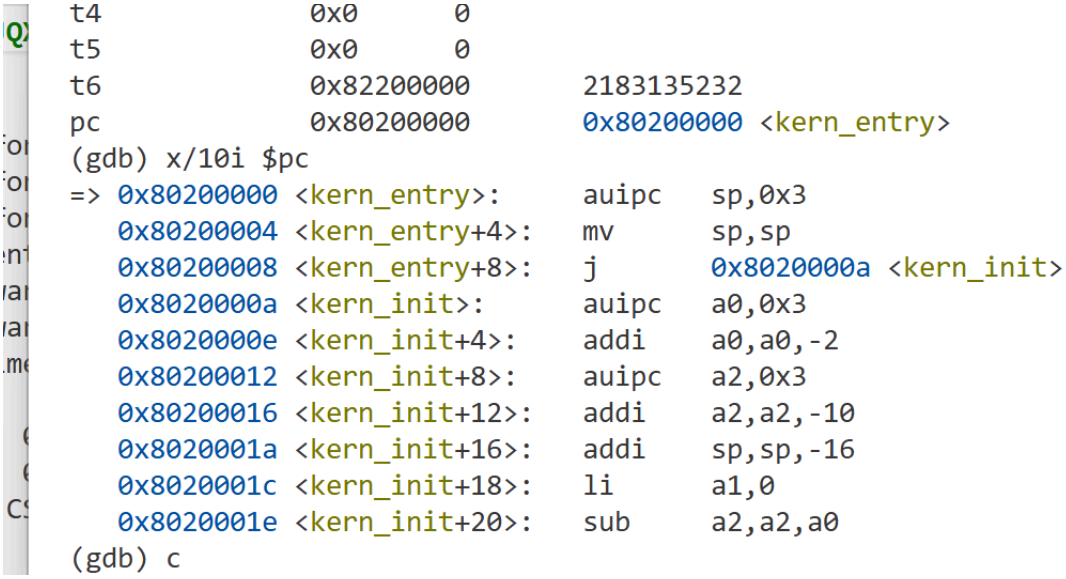
用 `b *0x80200000`指令设置断点, 继续运行至断点处,输出:

```
bash
Breakpoint 2, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
```

程序在断点处,即内核入口的位置暂停.

> 这里有一点是:我们已经知道qemu会将内核入口加载到0x80200000的位置,所以才直接通过地址设置断点

此时,再用 `info regiters`查看寄存器值, pc值已经变化为0x80200000.通过指令 `x/10i $pc`能够查看内核的前10条指令.

内核前10条指令	说明
	可以明显看出,内核的前10条指令实现地址偏移的计算,栈指针的初始化,跳转到c语言编写的内核初始函数等

继续输入指令c, 陷入无线循环,本次实验结束

练习问题回答

问题一回答

最初几条指令位于什么地址:OpenSBI的入口被映射/跳转挂在`0x1000`,所以在GDB第一条指令正位于`0x0000000000001000`,也就是复位向量地址.

功能:RISC-V处理器上电复位后，处理器从物理地址0x1000开始执行OpenSBI固件的机器模式初始化代码。这几条指令完成最基本的硬件环境建立：初始化a0、a1等寄存器以传递hartID和设备树地址，设置mstatus、mepc、mtvec寄存器并关闭中断，建立机器模式的栈指针，为各hart分配栈空间并让启动hart继续运行，同时配置PMP寄存器以开放内存访问权限。完成这些底层准备后，固件通过`jr`或`mret`指令跳转到位于0x80000000处的OpenSBI主体，负责后续加载并启动操作系统内核。

本节重要知识点

在本实验中，我们通过 QEMU + GDB 验证了 RISC-V 的启动流程及内核加载机制。复位向量地址位于 **0x1000** ，这是 QEMU virt 机器约定的 BootROM 起始地址。上电复位后，处理器的通用寄存器全部清零，而 **PC 寄存器会被硬件直接置为复位向量地址** ，因此执行的第一条指令位于 0x1000。此时执行的代码属于 OpenSBI 固件，用于完成机器模式的初始化和内核启动准备。

内核的入口地址固定在 **0x80200000** .当 OpenSBI 完成初始化后，通过寄存器 `mepc` 设定下一跳地址并执行 `mret`，控制权被移交到内核入口.GDB 断点验证显示，在到达 0x80200000 后，CPU 开始执行内核的启动代码 `entry.S`。该文件负责初始化内核栈指针（`la sp, bootstacktop`），然后通过 `tail kern_init` 将控制权交给 C 语言编写的 `kern_init` 函数，完成从汇编环境到 C 语言环境的切换.

因此，本节实验验证了 RISC-V 启动的关键过程： **从复位向量 → OpenSBI 初始化 → 跳转到内核入口 → 汇编启动代码 → C 内核初始化** 。

在本次实验中，我借助 QEMU 与 GDB 工具观察了 uCore 的启动过程, 巩固复习了理论课的知识.

首先,在 CPU 上电复位后,实验表明 PC 并非处于零值状态,而是被硬件强制装载为复位向量地址 `0x1000`.这与原理课程中"处理器从固件入口开始执行"的抽象描述相一致，而实验揭示了 RISC-V 在 QEMU 环境下的特定实现方式.

随后，实验中观测到 OpenSBI 固件完成了对硬件环境的基本搭建，包括 `mstatus`、`mepc`、`mtvec` 的配置，以及 hart ID 和设备树地址的传递.这一过程对应于原理课程中所强调的 BootLoader 功能，即在内核启动之前建立最小可行的执行环境.

在进入内核入口时，实验进一步揭示了 `entry.S` 中的关键动作.例如，通过 `la sp, bootstacktop` 设置栈指针，为后续 C 语言代码执行提供栈空间. 这一过程体现了从汇编世界过渡到 C 语言世界的桥梁作用.操作系统原理课程也会讲解函数调用和进程执行需要栈支持，但不会涉及如何在内核最初阶段完成栈的初始化。紧接着，`tail kern_init` 完成了控制权的不可返回移交，这与课程中所说的“从 BootLoader 到内核的单向控制权转移”在含义上完全一致，但实验中通过具体的伪指令表现出来，更具象、更贴近硬件.