Lab1

名称	版本信息
qemu	8.2.2
gcc	13.2.0
Virtual Machine	WSL2
OS	Ubuntu 24.04.3 LTS

我们先根据所给的代码来进行初步的实验的复现。在开始，输入命令`make qemu`，以下是输出的结果。

根据实验手册中的内容可知，最后的输出应该为`os is loading...`，但是图中的内容与理想的输出并不相符，我们通过查看图中的输出内容可知，`Domain0 Next Address : 0x0000000000000000`也就是说，在OpenSBI加载内核镜像的时的地址为`ox0`，无法正确加载内核。

在查询资料之后发现，`make qemu`命令只是加载二进制文件，但是二进制文件并没有ELF头，OpenSBI不能解析入口地址，因此`Domain0 Next Address`显示为0x0，导致跳转失败。

根据上述分析，我们选择解决方案为在`Makefile`文件中指定ELF格式的“内核文件”，使OpenSBI读取其ELF头，找到入口地址并跳转；同时这种解决方案无需更换qemu的版

本。进入Makefile文件中，原本的Makefile中的内容为

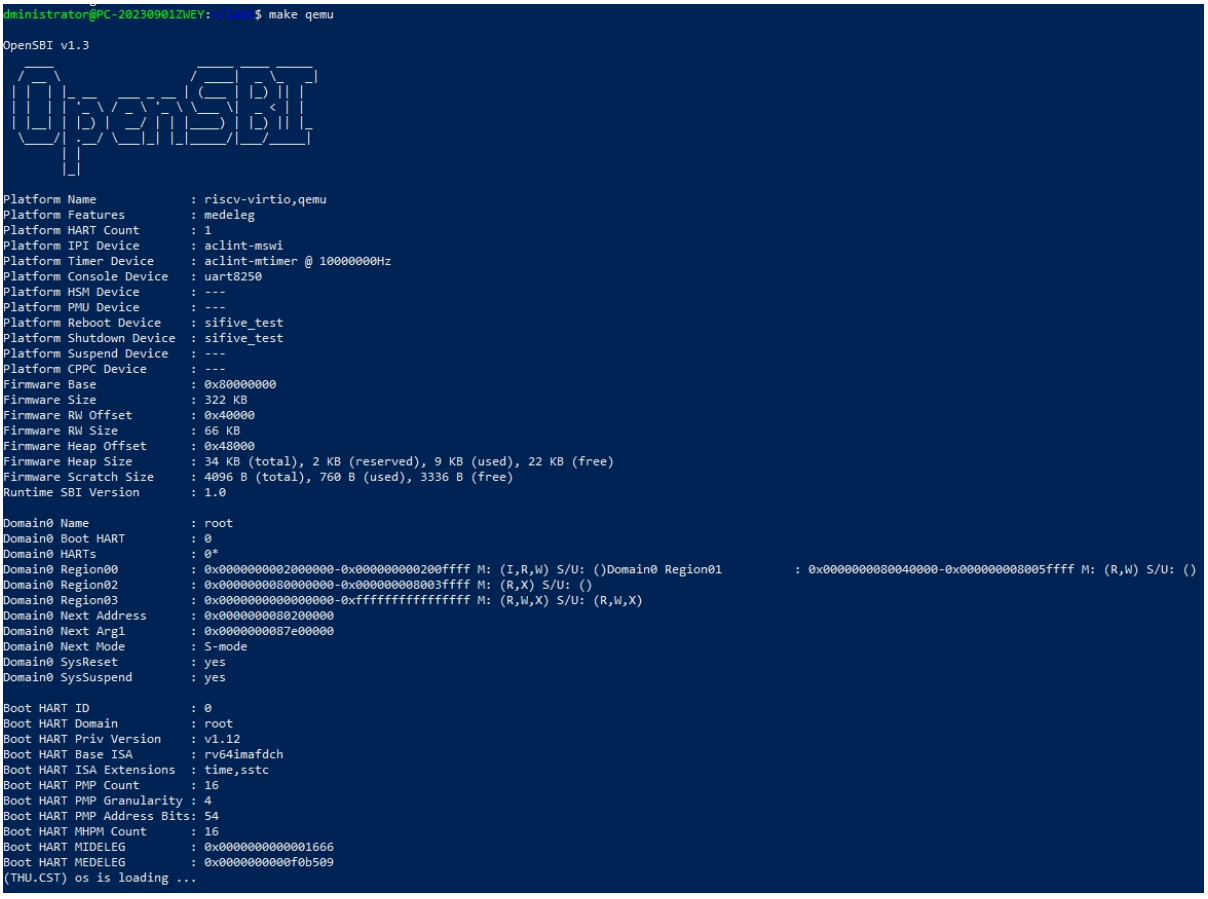
```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
        $(V)$(QEMU) \
                -machine virt \
                -nographic \
                -bios default \
                -device loader,file=$(UCOREIMG),addr=0x80200000
```

这个版本的问题是其使用了-device loader加载二进制镜像，导致OpenSBI无法解析入口地址，因此Domain0 Next Address=0，将原内容替换为：

```
.PHONY: qemu
qemu: bin/kernel
        @$(QEMU) -machine virt -nographic -bios default -kernel bin/kernel
```

通过传入ELF文件，OpenSBI能自动解析ELF header，从而得到正确入口地址，实现跳转执行，也就解决了Domain0 Next Address = 0x0的问题。

修改保存文件之后，再输入命令make qemu，得到的结果如下图所示：



根据结果可知，我们成功解决了Qemu加载内核方式不当的问题所导致的启动地址解析错误问题。

练习

练习一

先从`entry.S`文件本身入手，再来解答对应的两个问题。

`entry.S`文件中的代码是`RISC-V` 操作系统启动阶段的汇编引导代码，是内核执行的第一段代码。

1. `la sp,bootstacktop`：这一句代码是给内核设置一段栈空间，方便之后进入C函数调用。

`la`即`load address`，该指令会将`bootstacktio`符号对应的地址加载到寄存器`sp`中。`sp`是栈寄存器，RISC-V函数调用依赖它来管理栈空间。`bootstacktop`是内核启动时的内核栈顶地址。

2.`tail kern_init`：其中`tail`是RISC-V特有的尾调用指令，实际上是一个伪指令，其会跳转到`kern_init`处，但是`tail`的特别之处在于`tail`在跳转前会释放当前栈帧，不会保存返回地址到`ra`寄存器，相当于一个“不可返回的跳转”。也就是说，在执行完这一条指令之后，程序就直接进入`kern_init()`函数，不会再返回到`kern_entry`。

原因：`kern_entry`只是一个启动入口，不需要在`kern_init`执行完之后返回，同时使用`tail`而不是`call`，可以节省一次函数调用的开销，并符合操作系统引导阶段“一去不复返”的逻辑。

目的：总的来说，这一句是在内核启动的最初阶段，从启动入口`kern_entry`直接、不可返回地跳转到内核初始化函数`kern_init`，完成控制权的正式移交。

练习二

使用GDB验证启动流程

启动QEMU

输入命令

```
qemu-system-riscv64 -machine virt -nographic \
    -bios /usr/lib/riscv64-linux-gnu/opensbi/generic/fw_jump.elf \
    -kernel bin/kernel \
    -S -s
```

来启动QEMU，并加载OS内核，其中命令`-S`是QEMU启动后立即暂停CPU，不执行第一条指令，等待调试器，`-s`相当于`-gdb tcp::1234`，打开1234端口等待GDB连接。在输入该命令之后，QEMU会卡住，然后我们打开另一个终端，输入`riscv64-unknown-elf-gdb bin/kernel`命令启动调试，进入GDB之后，再输入以下命令连接QEMU：`target remote localhost:1234`。至此我们可以开始进行GDB调试。

GDB调试

```
dministrator@PC-20230901ZWEY:~/lab1$ riscv64-unknown-elf-gdb bin/kernel
GNU gdb (GDB) 16.3.90.20250610-git
Copyright (C) 2024 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 "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".
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:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/bugs/>.
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
```

在输入远程连接的命令之后，终端输入的内容为：

```
0x0000000000001000 in ?? ()
```

这表示当前CPU的程序计数器（PC）的值为`0x1000`，但GDB并不知道这段地址对应的符号，所以显示`in ??()`。这一段内容正好印证了RISC-V在加电复位后，PC会指向`0x1000`，也就是OpenSBI固件的入口地址。因为我们调试的对象是裸机代码/固件，不是普通ELF带符号的应用程序，GDB对0x1000这个地址没有调试符号，所以只显示地址

而不显示函数名。同时，也表明了我们已经把GDB和QENU“接好线”了，当前CPU“停在”地址0x1000，还没开始执行内核初始化。

接下来输入`x/10i 0x1000`查看前十条汇编语句，结果如下图所示：

```
(gdb) x/10i 0x1000
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a2,t0,40
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      a1,32(t0)
   0x1010:      ld      t0,24(t0)
   0x1014:      jr      t0
   0x1018:      unimp
   0x101a:      .insn   2, 0x8000
   0x101c:      unimp
   0x101e:      unimp
(gdb)
```

然后输入命令`info registers`来查看寄存器中的内容，结果如下图所示：

```
ra             0x0      0x0
sp             0x0      0x0
gp             0x0      0x0
tp             0x0      0x0
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x0      0x0
s1             0x0      0
a0             0x0      0
a1             0x0      0
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7             0x0      0
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
s8             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--c
s9             0x0      0
s10            0x0      0
s11            0x0      0
t3             0x0      0
t4             0x0      0
t5             0x0      0
t6             0x0      0
pc             0x1000   0x1000
```

在未运行时，通用寄存器的值大多为0，这表明系统处于干净的初始化状态，但是PC寄存器的值为0x1000，这是当前正在执行的指令地址，同时也是监控模式代码的执行位置，系统准备切换到监管模式。

我们设置内核入口断点`break *0x80200000`来观察内核加载瞬间，结果如图：

```
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb)
```

该处得到的结果与我们在练习一中的分析是一致的。使用命令`continue`来运行至我们设置的断点处。

之后，我们再使用命令`info registers`来查看此时的寄存器的内容：

```
ra             0x8000b962       0x8000b962
sp             0x80046eb0       0x80046eb0
gp             0x0      0x0
tp             0x80047000       0x80047000
t0             0x0      0
t1             0x2      2
t2             0x1000   4096
fp             0x80046ef0       0x80046ef0
s1             0x1      1
a0             0x0      0
a1             0x82200000       2183135232
a2             0x7      7
a3             0x19     25
a4             0x80     128
a5             0x1      1
a6             0x1      1
a7             0x5      5
s2             0x0      0
s3             0x80200000       2149580800
s4             0x0      0
s5             0x82200000       2183135232
s6             0x8000000a00006800       -9223371993905076224
s7             0x7f     127
s8             0x2000   8192
--Type <RET> for more, q to quit, c to continue without paging--c
s9             0x800436f0       2147759856
s10            0x0      0
s11            0x0      0
t3             0x9      9
t4             0x80046ef0       2147774192
t5             0x100000000      4294967296
t6             0x20     32
pc             0x80200000       0x80200000 <kern_entry>
```

从这些寄存器中的内容可知，系统已经完成了初始化，正在正常执行代码。其中PC的值为0x80200000，表明这时正在执行地址0x80200000处的代码，同时这个位置也被标记为kern_entry即内核入口点，系统刚完成从OpenSBI到内核的控制权转移。

此时寄存器的特点为： 1.**控制权刚转移：** PC指向内核入口，但其他寄存器还保留着OpenSBI的设置

2.**混合环境：** 栈指针还在OpenSBI区域，但代码执行已进入内核空间。

3.**参数传递：** OpenSBI通过a0-a7寄存器向内核传递启动信息

4.**设备信息：** a1寄存器包含设备树地址，这是内核了解硬件配置的关键。

再输入命令`x/10i $pc`来查看此时的前10条汇编语句，结果如下图所示：

```
(gdb) x/10i $pc
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
   0x80200004 <kern_entry+4>:   mv      sp,sp
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
   0x8020000a <kern_init>:      auipc   a0,0x3
   0x8020000e <kern_init+4>:    addi    a0,a0,-2
   0x80200012 <kern_init+8>:    auipc   a2,0x3
   0x80200016 <kern_init+12>:   addi    a2,a2,-10
   0x8020001a <kern_init+16>:   addi    sp,sp,-16
   0x8020001c <kern_init+18>:   li      a1,0
   0x8020001e <kern_init+20>:   sub     a2,a2,a0
(gdb)
```

以上便是本次实验的全部内容。

解答练习问题

RISC-V 硬件加电后最初执行的几条指令位于什么地址： OpenSBI的入口被映射在`0x1000`，所以GDB的第一条指令的地址为：`0x0000000000001000`，也就是复位向量。之后的指令就在此基础上依次加4位。

功能： 在RISC-V硬件加电后最初的几条指令的主要功能是进行最基本的CPU初始化，并跳转到OpenSBI的主初始化函数，完成了从复位向量到C语言运行环境的桥梁作用。

以前10条指令为例：

```
0x1000:    auipc   t0, 0x0 ;将当前PC值（0x1000）存入t0
0x1004:    addi    a2, t0, 40 ; 将t0的值加上40，结果存入a2
0x1008:    csrr    a0, mhartid ;读取 mhartid CSR（控制和状态寄存器）的值到a0
0x100c:    ld      a1, 32(t0) ;从地址 t0 + 32 处加载一个双字到a1
0x1010:    ld      t0, 24(t0) ;从地址 t0 + 24 处加载一个双字到t0
0x1014:    jr      t0 ;跳转到t0寄存器指定的地址
0x1018:    unimp ;未实现指令和指令填充
0x101a:    .insn   2, 0x8000
0x101c:    unimp
0x101e:    unimp
```

本次实验重要知识点

1.启动流程与复位向量

复位向量地址0x1000，PC上电后被硬件置为该地址，在实验过程中，通过查看PC的值验证了这一点，即“处理器从固件入口开始执行”。

2.BootLoader功能

OpenSBI完成机器模式初始化、寄存器配置、内核启动准备，通过查看RISC-V 硬件加电后最初执行的指令可以观测到mstatus、mepc、mtvec配置，hart ID传递。

3.内核入口与控制权转移

内核入口0x80200000，通过mret指令移交控制权。

4.运行环境初始化

entry.S文件设置栈指针，从汇编过渡到C语言，为操作系统运行准备环境，对应着`la sp, bootstacktop`，通过该句命令来具体的实现栈初始化。

本次实验未涉及的重要知识点

在本次实验中，我们主要实现了最小可执行内核的构建和系统启动流程的验证，尚未涉及操作系统核心功能的完整实现。具体而言，以下关键知识点暂未包含在本次实验范围内：

进程管理与调度方面，实验尚未实现进程控制块、进程状态转换、上下文切换机制，也未涉及任何调度算法（如时间片轮转、优先级调度等）。

内存管理相关功能仍待开发，包括虚拟内存机制、多级页表管理、地址空间转换、TLB优化以及页面置换算法等重要内容。

文件系统的实现尚未展开，实验未包含文件与目录的抽象模型、inode数据结构、磁盘空间分配策略及相关的调度算法。

设备管理与系统调用机制同样不在本实验范围内，包括设备驱动程序框架、中断处理流程、用户态与内核态的切换等关键功能。

这些核心操作系统组件将在后续实验中逐步构建和完善，形成完整的操作系统功能体系。