Lab1 实验报告

组员:马思远 石爽 薄照轩

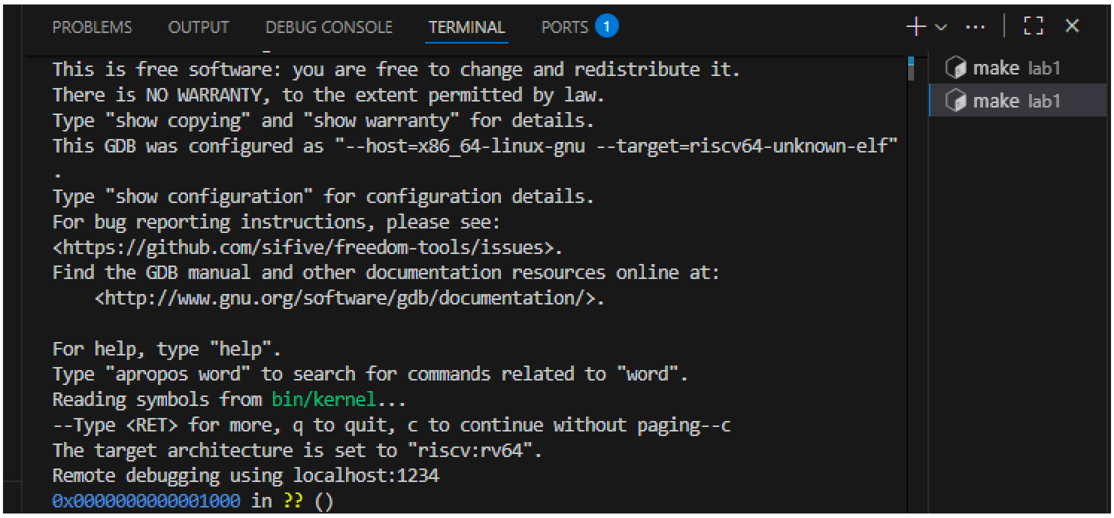
练习 1：理解内核启动中的程序入口操作

la sp, bootstacktop 会把 bootstacktop 的地址(即内核栈的栈顶地址)存入 sp 寄存器。目的是为后续的 C 语言函数调用准备“内核栈”,为栈分配内存空间.

tail 是 RISC-V 汇编中的“尾调用”指令，作用是跳转到 kern_init 函数，并让 kern_init 的返回地址被忽略.目的是进入 c 语言编写的内核入口函数 kern_init. Entry.s 是内核的汇编入口,kern_init 则是 c 语言入口,tail 完成了汇编到 c 语言的交接

练习 2: 使用 GDB 验证启动流程

打开两个终端,分别执行 make debug 和 make gdb,观察终端 GDB 停在 0x0000000000001000 in ?? ()

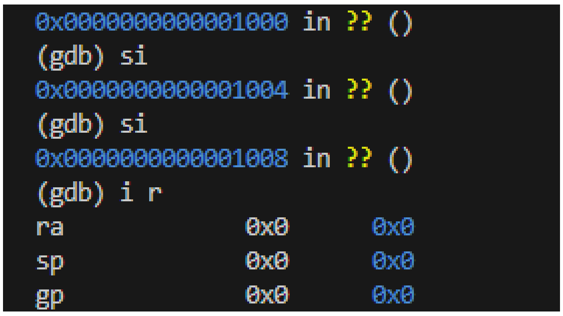


```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf"
.
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
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 bin/kernel...
--Type <RET> for more, q to quit, c to continue without paging--c
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
```

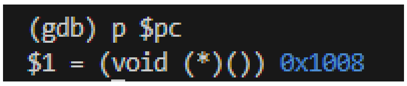
这里其实是 QEMU 内置的固件（BIOS）代码，还没执行到我们的内核。

让它执行两次



```
0x0000000000001000 in ?? ()
(gdb) si
0x0000000000001004 in ?? ()
(gdb) si
0x0000000000001008 in ?? ()
(gdb) i r
ra             0x0      0x0
sp             0x0      0x0
gp             0x0      0x0
```

观察 pc 值

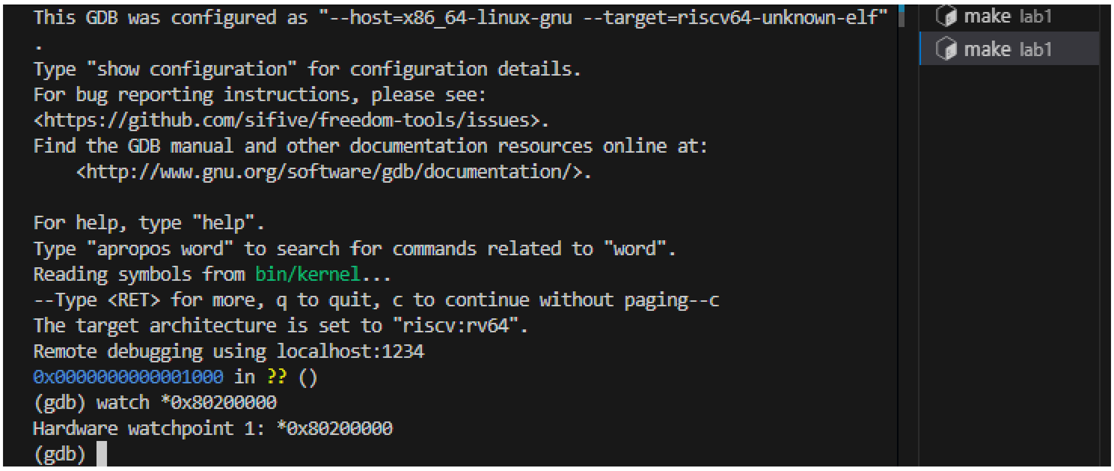


```
(gdb) p $pc
$1 = (void (*)()) 0x1008
```

推算得

RISC-V 加电后最初执行的指令位于 0x1000 说明 CPU 从复位地址（0x1000）开始执行初始化固件（OpenSBI）的汇编代码，进行最基础的硬件初始化。

我们用命令 watch *0x80200000 观察内核加载

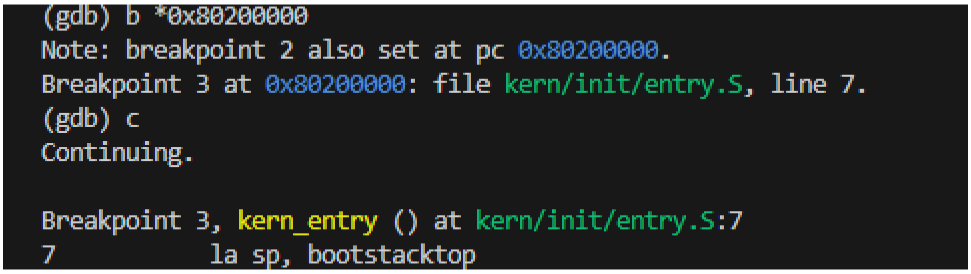


```
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf"
.
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...
--Type <RET> for more, q to quit, c to continue without paging--c
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) watch *0x80200000
Hardware watchpoint 1: *0x80200000
(gdb) 
```

GDB 会提示 Hardware watchpoint 1: *0x80200000，表示已设置“当 0x80200000 地址的内存被写入时，自动中断”；观察到内核加载瞬间

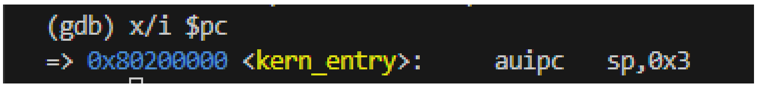
之后在物理地址 0x80200000 处设置了一个断点。GDB 提示“Breakpoint 3 at 0x80200000: file kern/init/entry.S, line 7”，说明该断点对应到了 kern/init/entry.S 文件的第 7 行代码，也就是内核汇编入口 kern_entry 函数的起始位置



```
(gdb) b *0x80200000
Note: breakpoint 2 also set at pc 0x80200000.
Breakpoint 3 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) c
Continuing.

Breakpoint 3, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
```

使用 x/i $pc 命令查看当前 pc（程序计数器）指向的指令，输出为 auipc sp,0x3。auipc 是 RISC-V 架构中的指令，作用是将立即数的高 20 位与当前 pc 的高 44 位拼接，结果存入目标寄存器（这里是 sp，栈指针寄存器）。结合 entry.S 的代码逻辑，这一步是在为内核栈的设置做准备，通过 auipc 指令结合后续的偏移计算，最终将栈指针 sp 指向 bootstacktop（内核栈的栈顶），为后续的 C 语言函数调用（如 kern_init）准备好栈环境。



```
(gdb) x/i $pc
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
```

综上，这一系列调试操作验证了从 OpenSBI 移交控制权到内核入口 kern_entry 执行的过程

RISC-V 硬件加电后，pc 寄存器的初始值为 0x1000，即最初执行的几条指令位于物理地址 0x1000,这些代码的作用:

1.对 CPU 核心寄存器、内存控制器等最基础硬件进行复位和初始化，确保后续程序能正常访问内存和设备

2.引导 Bootloader：将控制权转移到更完善的固件（OpenSBI）