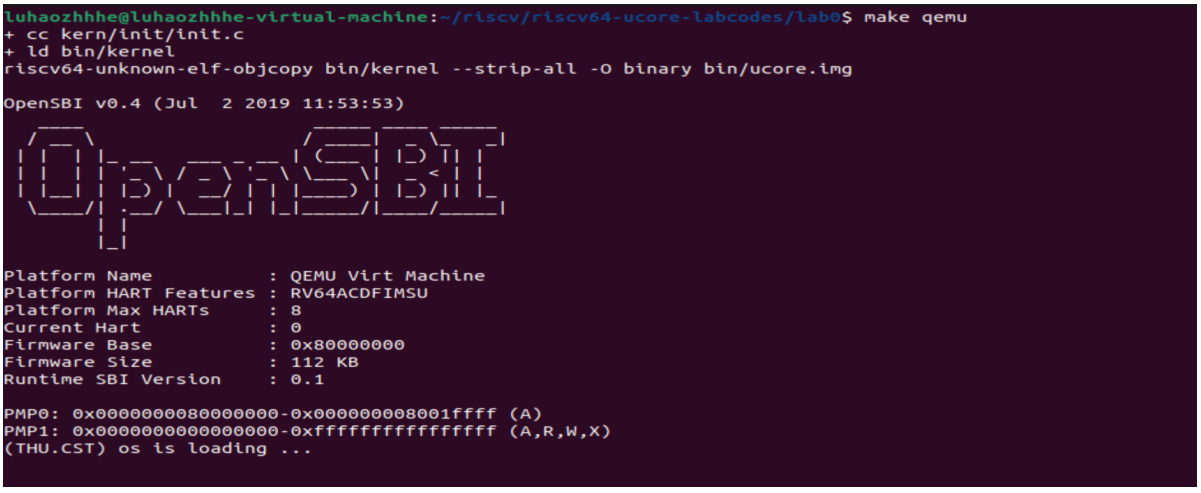
Lab0.5

首先测试make qemu的语句，运行得到：



说明我们的makefile是正确的

然后下面开始做我们的调试工作

一、实验过程

使用`make gdb`调试，输入指令`x/10i $pc`查看即将执行的10条汇编指令，其中在地址为`0x1010`的指令处会跳转，故实际执行的为以下指令：

```
0x1000:      auipc   t0,0x0      # t0 = pc + 0 << 12 = 0x1000
0x1004:      addi    a1,t0,32    # a1 = t0 + 32 = 0x1020
0x1008:      csrr    a0,mhartid # a0 = mhartid = 0
0x100c:      ld      t0,24(t0)  # t0 = [t0 + 24] = 0x80000000
0x1010:      jr      t0         # 跳转到地址0x80000000
```

输入`si`单步执行，使用形如`info r t0`的指令查看涉及到的寄存器结果：

```
(gdb) si
0x0000000000001004 in ?? ()
(gdb) info r t0
t0             0x0000000000001000   4096
(gdb) si
0x0000000000001008 in ?? ()
(gdb) info r t0
t0             0x0000000000001000   4096
(gdb) si
0x000000000000100c in ?? ()
(gdb) info r t0
t0             0x0000000000001000   4096
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0             0x0000000080000000   2147483648  #由于在上一步的执行过程中t0 = [t0 +
24] = 0x80000000，值变化了，所以t0寄存器里的值也改变了
(gdb) si
```

```
0x0000000080000000 in ?? ()                    #跳转到地址0x80000000
```

之后会跳转到地址`0x80000000`处继续执行，输入x/10i 0x80000000，显示0x80000000处的10条数据。该地址处加载的是作为bootloader的`OpenSBI.bin`，该处的作用为加载操作系统内核并启动操作系统的执行。代码如下：

```
0x80000000: csrr a6,mhartid                 # a6 = mhartid (获取当前硬件线程的ID)
0x80000004: bgtz a6,0x80000108              # 如果 a6 > 0，则跳转到0x80000108
0x80000008: auipc t0,0x0                   # t0 = pc + (0x0 << 12) = 0x80000008
0x8000000c: addi t0,t0,1032                # t0 = t0 + 1032 = 0x80000408
0x80000010: auipc t1,0x0                   # t1 = pc + (0x0 << 12) = 0x80000010
0x80000014: addi t1,t1,-16                 # t1 = t1 - 16 = 0x80000000
0x80000018: sd t1,0(t0)                    # 将t1的值（0x80000000）存储在地址0x80000408
处
0x8000001c: auipc t0,0x0                   # t0 = pc + (0x0 << 12) = 0x8000001c
0x80000020: addi t0,t0,1020                # t0 = t0 + 1020 = 0x80000400
0x80000024: ld t0,0(t0)                    # t0 = [t0 + 0] = [0x80000400] (从地址
0x80000400加载一个双字到t0)
```

接着输入指令`break kern_entry`，在目标函数kern_entry的第一条指令处设置断点，输出如下：

```
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

地址`0x80200000`由`kernel.ld`中定义的`BASE_ADDRESS`（加载地址）所决定，标签`kern_entry`是在`kernel.ld`中定义的`ENTRY`（入口点）

`kernel_entry`标志的汇编代码及解释如下：

- `la sp, bootstacktop`：将`bootstacktop`的地址赋给`sp`，作为栈
- `tail kern_init`：尾调用，调用函数`kern_init`

输入指令`x/5i 0x80200000`，查看汇编代码：

```
0x80200000 <kern_entry>:      auipc   sp,0x3          # sp = pc + (0x3 << 12) =
0x80200000 + (0x3 << 12) = 0x80203000
0x80200004 <kern_entry+4>:    mv      sp,sp           # sp = sp (这条指令实际上没有
改变sp的值，可能是为了某些同步/延迟原因)
0x80200008 <kern_entry+8>:    j       0x8020000c <kern_init> # 无条件跳转到地址
0x8020000c
0x8020000c <kern_init>:       auipc   a0,0x3          # a0 = pc + (0x3 << 12) =
0x8020000c + (0x3 << 12) = 0x8020300c
0x80200010 <kern_init+4>:     addi    a0,a0,-4        # a0 = a0 - 4 = 0x8020300c
- 4 = 0x80203008
```

可以看到在`kern_entry`之后，紧接着就是`kern_init`

输入`continue`执行直到断点，debug输出如下：

```
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)
```

这说明OpenSBI此时已经启动。

接着输入指令`break kern_init`，输出如下：

```
Breakpoint 2 at 0x8020000c: file kern/init/init.c, line 8.
```

这里就指向了之前显示为`<kern_init>`的地址`0x8020000c`

> 补充一些寄存器：
>
> - `ra`：返回地址
> - `sp`：栈指针
> - `gp`：全局指针
> - `tp`：线程指针

输入`continue`，接着输入`disassemble kern_init`查看反汇编代码：

```
0x0000000080200000c <+0>:    auipc   a0,0x3          # a0 = pc + (0x3 << 12)，即a0 = 0x8020000c + 0x3000 = 0x8020300c
0x0000000080200010 <+4>:    addi    a0,a0,-4        # a0 = a0 - 4，即a0 = 0x8020300c - 4 = 0x80203008
0x0000000080200014 <+8>:    auipc   a2,0x3          # a2 = pc + (0x3 << 12)，即a2 = 0x80200014 + 0x3000 = 0x80203014
0x0000000080200018 <+12>:   addi    a2,a2,-12       # a2 = a2 - 12，即a2 = 0x80203014 - 12 = 0x80203002
0x000000008020001c <+16>:   addi    sp,sp,-16       # sp = sp - 16 (在堆栈上分配16字节的空间)
0x000000008020001e <+18>:   li      a1,0            # a1 = 0 (立即加载0到a1寄存器)
0x0000000080200020 <+20>:   sub     a2,a2,a0        # a2 = a2 - a0，即a2 = 0x80203002 - 0x80203008 = -6
0x0000000080200022 <+22>:   sd      ra,8(sp)        # 将返回地址(ra)存储到堆栈的sp+8位置
0x0000000080200024 <+24>:   jal     ra,0x802004ce <memset>  # 跳转到memset函数，并设置返回地址(ra)
0x0000000080200028 <+28>:   auipc   a1,0x0          # a1 = pc + (0x0 << 12)，即a1 = 0x80200028
```

```
0x0000000080200028 <+32>:    addi    a1,a1,1208      # a1 = a1 + 1208，即a1 = 0x80200028 + 1208 = 0x802004e0
0x0000000080200030 <+36>:    auipc   a0,0x0          # a0 = pc + (0x0 << 12)，即a0 = 0x80200030
0x0000000080200034 <+40>:    addi    a0,a0,1232      # a0 = a0 + 1232，即a0 = 0x80200030 + 1232 = 0x80200500
0x0000000080200038 <+44>:    jal     ra,0x80200058 <cprintf> # 跳转到cprintf函数，并设置返回地址(ra)
0x000000008020003c <+48>:    j       0x8020003c <kern_init+48>  # 跳转到地址0x8020003c处的指令
```

可以看到这个函数最后一个指令是`j 0x8020003c <kern_init+48>`，也就是跳转到自己，所以代码会在这里一直循环下去。

输入`continue`，debug窗口出现以下输出：

```
(THU.CST) os is loading ...
```

二、练习1回答

1. RISCV加电后的指令在地址`0x1000`到地址`0x1010`。
2. 完成的功能如下：
    - `auipc t0,0x0`：用于加载一个20bit的立即数，`t0`中保存的数据是`(pc)+(0<<12)`。用于PC相对寻址。
    - `addi a1,t0,32`：将`t0`加上`32`，赋值给`a1`。
    - `csrr a0,mhartid`：读取状态寄存器`mhartid`，存入`a0`中。`mhartid`为正在运行代码的硬件线程的整数ID。
    - `ld t0,24(t0)`：双字，加载从`t0+24`地址处读取8个字节，存入`t0`。
    - `jr t0`：寄存器跳转，跳转到寄存器指向的地址处（此处为`0x80000000`）。