

操作系统lab1

小组成员：苏奕扬、姚鑫秋、侯懿轩

分工：在三个组员对所有内容熟练掌握的基础上，苏奕扬负责编写练习1部分报告，侯懿轩负责练习二第一部分，姚鑫秋负责练习二第二三部分。

练习一：理解内核启动中的程序入口操作

1. `la sp,bootstacktop`：la 是 Load Address（伪指令），用于把一个符号（如全局变量或标签）的地址加载到寄存器中。这条指令表示将 `bootstacktop` 的地址加载到栈指针寄存器 `sp` 中，目的是初始化栈，为栈分配内存空间。

```
(gdb) i r
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
--Type <RET> for more, q to quit, c to continue without p
aging--b* kern_entryRET
s3          0x0      0
s4          0x0      0
s5          0x0      0
s6          0x0      0
s7          0x0      0
s8          0x0      0
s9          0x0      0
s10         0x0      0
s11         0x0      0
t3          0x0      0
t4          0x0      0
t5          0x0      0
t6          0x0      0
pc          0x1000   0x1000
```

```

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) i r
ra          0x80000a02      0x80000a02
sp          0x8001bd80      0x8001bd80
gp          0x0            0x0
tp          0x8001be00      0x8001be00
t0          0x80200000      2149580800
t1          0x1            1
t2          0x1            1
fp          0x8001bd90      0x8001bd90
s1          0x8001be00      2147597824
a0          0x0            0
a1          0x82200000      2183135232
a2          0x80200000      2149580800
a3          0x1            1
a4          0x800          2048
a5          0x1            1
a6          0x82200000      2183135232
a7          0x80200000      2149580800
s2          0x800095c0      2147521984
s3          0x0            0
s4          0x0            0
s5          0x0            0
s6          0x0            0
s7          0x8            8
s8          0x2000          8192
s9          0x0            0
s10         0x0            0
s11         0x0            0
t3          0x0            0
t4          0x0            0
t5          0x0            0
t6          0x82200000      2183135232
pc          0x80200000      0x80200000 <kern_entry>

```

由图示可以看出，sp寄存器被赋值为0x8001bd80。

2. `tail kern_init`：无返回地跳转到 `kern_init` 函数执行，完成各类硬件和内存初始化，启动操作系统核心。

```

9          tail kern_init
(gdb) si
kern_init () at kern/init/init.c:8
8          memset(edata, 0, end - edata);

```

如图所示，在`tail kern_init`指令后进行`si`单步调试，程序跳转到`init.c`文件中的`kern_init`函数进行执行。

练习二：使用GDB验证启动流程

调试过程记录与分析

本次调试的目标是利用QEMU和GDB，精确跟踪并验证RISC-V平台从硬件加电到操作系统内核获得控制权的完整启动链条。为此，我设计并执行了如下的分阶段调试策略：

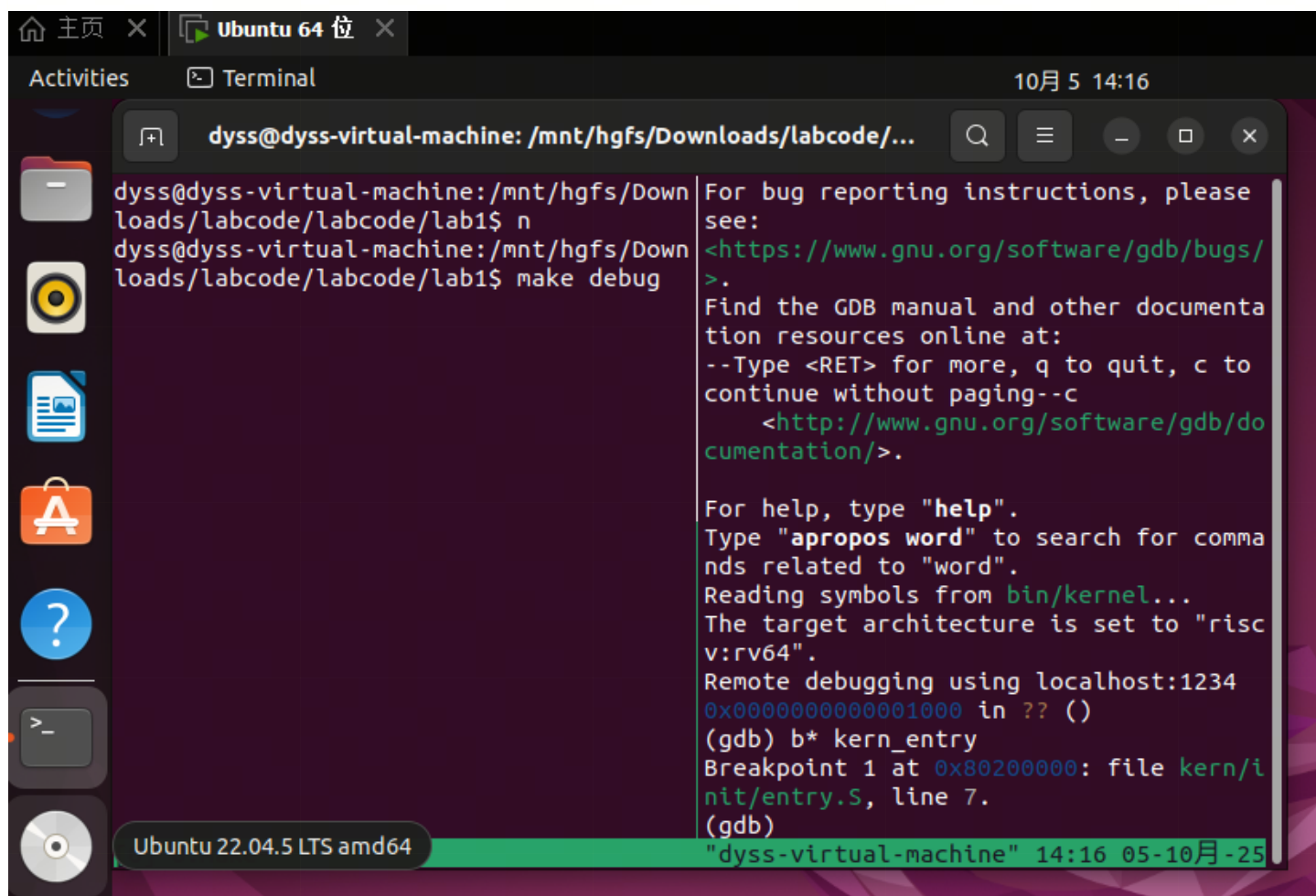
阶段一：硬件复位瞬间

首先，为了验证CPU的起始执行点，我通过 `make debug` 和 `make gdb` 启动了远程调试会话。连接成功后，GDB准确地停在了 `0x00000000000001000 in ?? ()`，这直接验证了QEMU模拟的RISC-V `virt` 机器的**硬件复位地址（Reset Vector）**为 `0x1000`。

通过 `x/5i $pc` 指令反汇编此处的代码，可以观察到一段固化在ROM中的底层复位代码。这段代码的功能极为专一：它进行最基础的CPU状态设置，其最终目的是执行一条跳转指令，将控制权无条件移交给下一阶段的固件——OpenSBI。

我们的操作步骤为：

首先，我们根据指导书使用 `tmux` 进行分屏操作，在左边窗口进行 `make debug`，在右边窗口进行 `make gdb`，然后出现了以下内容：



```
dyss@dyss-virtual-machine: /mnt/hgfs/Downloads/labcode/...
dyss@dyss-virtual-machine:/mnt/hgfs/Downloads/labcode/labcode/lab1$ n
dyss@dyss-virtual-machine:/mnt/hgfs/Downloads/labcode/labcode/lab1$ make debug

For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
--Type <RET> for more, q to quit, c to continue without paging--c
<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...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x00000000000001000 in ?? ()
(gdb) b* kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb)
```

我们可以看到，cpu加电后，会直接让PC被硬件设置为一个固定地址，就是 `0x1000`。

`0x1000` 处存放的是固化在模拟ROM中的**复位向量代码（Reset Vector Code）**

然后，我们使用 `(gdb) x/10i 0x1000` 来观察在此地址上有什么指令：

```
dyss@dyss-virtual-machine: /mnt/hgfs/Downloads/labcode/...
dyss@dyss-virtual-machine:/mnt/hgfs/Down
loads/labcode/labcode/lab1$ m
ake debug
qemu-system-riscv64: -s: Failed to find
an available port: Address already in us
e
make: *** [Makefile:169: debug] Error 1
dyss@dyss-virtual-machine:/mnt/hgfs/Down
loads/labcode/labcode/lab1$ make debug
qemu-system-riscv64: -s: Failed to find
an available port: Address already in us
e
make: *** [Makefile:169: debug] Error 1
dyss@dyss-virtual-machine:/mnt/hgfs/Down
loads/labcode/labcode/lab1$ pkill -f qem
u
dyss@dyss-virtual-machine:/mnt/hgfs/Down
loads/labcode/labcode/lab1$ make debug

<http://www.gnu.org/software/gdb/do
cumentation/>.

For help, type "help".
Type "apropos word" to search for comma
nds related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "risc
v:rv64".
Remote debugging using localhost:1234
0x00000000000001000 in ?? ()
(gdb) x/10i $pc
=> 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) [1] 0:make*
```

现在让我来详细解释一下每一条指令是在干什么事情。

1. `auipc t0, 0x0`：这条指令是让pc值的高 20 位与立即数 `0x0` 相加，结果存入 t0 寄存器，即 $t0 = PC + 0x000000 = 0x1000$
2. `addi a2, t0, 40`：将 t0 的值加上 40，结果存入 a2； $a2 = 0x1000 + 40 = 0x1028$

我们在 `0x1000` 处进行 `si`（逐步执行）操作，然后使用 `info registers t0, a0,a1,a2` 来观察寄存器状态，然后来验证指令是否正确执行。下图我们可以看到a2被正确存入 `0x1028` 这个地址。

```
(gdb) si
0x00000000000001008 in ?? ()
(gdb) info registers t0 a0 a1 a2
t0          0x1000      4096
a0          0x0        0
a1          0x87000000   2264924
160
a2          0x1028     4136
(gdb)
```

1. `csrr a0, mhartid`：读取 `mhartid` CSR（控制和状态寄存器）到 a0
2. `ld a1, 32(t0)`：从内存地址 $t0 + 32 = 0x1000 + 32 = 0x1020$ 加载数据到 a1
3. `ld t0, 24(t0)`：从内存地址 $t0 + 24 = 0x1000 + 24 = 0x1018$ 加载数据到 t0
4. `jr t0`：跳转到 t0 寄存器指定的地址：跳转到 OpenSBI 的主初始化函数。

```
(gdb) si
0x00000000000001014 in ?? ()
(gdb) info registers t0 a0 a1 a2
t0                0x80000000          2147483
648
a0                0x0                0
a1                0x87000000          2264924
160
a2                0x1028            4136
(gdb)
```

我们可以看到，t0的地址变为 `0x80000000`，这里就是跳转到 `OpenSBI` 的主初始化函数的契机！
我们跳转到 `0x80000000` 这个地址去看一下具体的汇编指令：

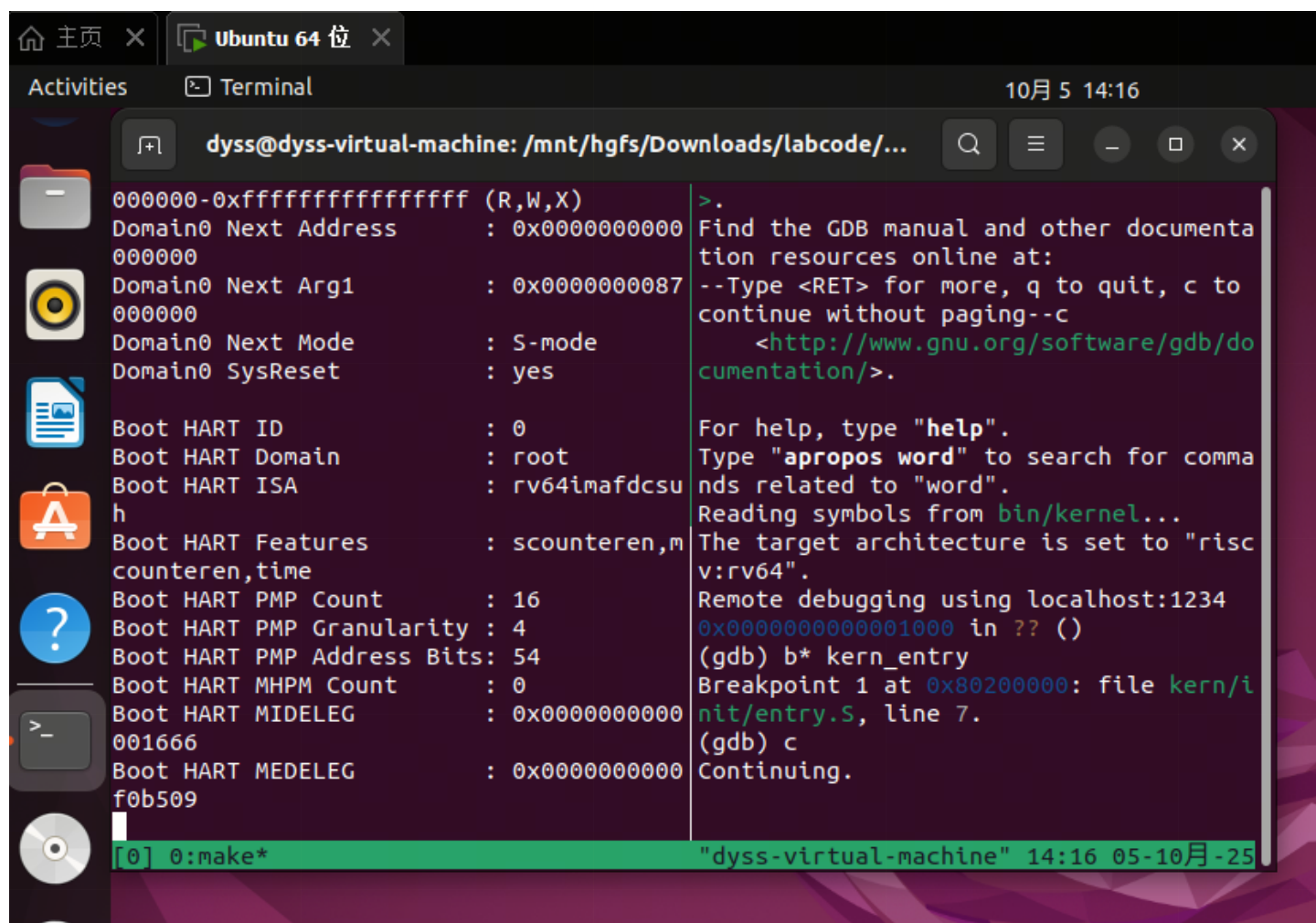
```
dyss@dyss-virtual-machine: /mnt/hgfs/Downloads/labcode/...
dyss@dyss-virtual-machine:/mnt/hgfs/Down
loads/labcode/labcode/lab1$ m
ake debug
Find the GDB manual and other documenta
tion resources online at:
--Type <RET> for more, q to quit, c to
continue without paging--c
<http://www.gnu.org/software/gdb/do
cumentation/>.

For help, type "help".
Type "apropos word" to search for comma
nds related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "risc
v:rv64".
Remote debugging using localhost:1234
0x00000000000001000 in ?? ()
(gdb) set $pc = 0x80000000
(gdb) x/5i $pc
=> 0x80000000: add    s0,a0,zero
0x80000004: add    s1,a1,zero
0x80000008: add    s2,a2,zero
0x8000000c: jal    0x80000560
0x80000010: add    a6,a0,zero
(gdb)
```

我们可以看到，这个代码已经是OpenSBI的主初始化代码，它会通过一系列的跳转代码，最后到内核的入口。但是追踪它看它何时能到内核入口是个很漫长的过程，于是我们直接借助链接脚本的力量，它指定了内核的入口地址。我们可以直接在这个地址上打断点，但更简单的方法是使用函数名。因为编译器帮我们把函数名和地址对应了起来（调试符号），所以我们可以直接对 `kern_entry` 函数下断点（`b` 是 `break` 的缩写）：

```
(gdb) b* kern_entry
```

然后输入 `c` 来运行，内核会在这个函数的入口停止。



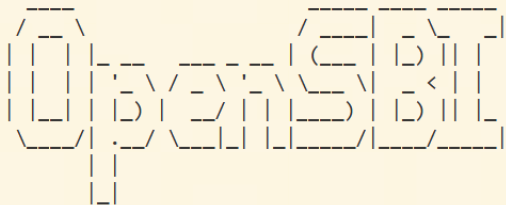
我们可以看到，内核的入口是： `0x80200000`，然后我们使用 `watch *0x80200000` 观察内核加载瞬间。

阶段二：观察内核加载并思考出现的问题

根据 `tips` 的指导，我使用 `watch *0x80200000` 观察内核加载瞬间，但是却得到了下面的结果。


```
yqqos@yxq:~/labcode/lab1$ make debug
```

```
OpenSBI v0.9
```



```
Platform Name      : riscv-virtio,qemu
Platform Features  : timer,mfdeleg
Platform HART Count : 1
Firmware Base      : 0x80000000
Firmware Size      : 100 KB
Runtime SBI Version : 0.2
```

```
Domain0 Name       : root
Domain0 Boot HART   : 0
Domain0 HARTs       : 0*
Domain0 Region00    : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01    : 0x0000000000000000-0xffffffff (R,W,X)
Domain0 Next Address : 0x0000000080200000
Domain0 Next Arg1    : 0x0000000087000000
Domain0 Next Mode    : S-mode
Domain0 SysReset     : yes
```

```
Boot HART ID       : 0
Boot HART Domain    : root
Boot HART ISA       : rv64imafdcsv
Boot HART Features   : scounteren,mcounteren,time
Boot HART PMP Count  : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG    : 0x0000000000000222
Boot HART MEDELEG    : 0x000000000000b109
(THU.CST) os is loading ...
```

```
yqqos@yxq:~/labcode/lab1$ make gdb
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
```

```
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
```

```
Copyright (C) 2020 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-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>.
```

```
< quit, c to continue without paging--c
```

```
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...
```

```
The target architecture is set to "riscv:rv64".
```

```
Remote debugging using localhost:1234
```

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

```
(gdb) watch *0x80200000
```

```
Hardware watchpoint 1: *0x80200000
```

```
(gdb) c
```

```
Continuing.
```

陷入了循环，并没有触发观察点。阅一些资料发现，真实硬件情况下，在输入watch *0x80200000后，会在内存中设置一个观察点，当输入c运行后，Bootloader会初始化硬盘并把内核文件读取到内存中并触发观察点，但是在QEMU中，QEMU为了简化流程，在启动时就已经预先将内核镜像放置在了0x80200000的内存处，所以就不会触发观察点。下面我来验证一下

我开启了两个新的窗口，在硬件复位后，我直接输入 `x/10i 0x80200000`，发现已经进入汇编入口点 (`kern/init/entry.S`)。证明我们的猜测是正确的，所以直接进行下一步。

```
yqqos@yxq:~/labcode/lab1$ make debug
```

```
yqqos@yxq:~/labcode/lab1$ make gdb
riscv64-unknown-elf-gdb \
  -ex 'file bin/kernel' \
  -ex 'set arch riscv:rv64' \
  -ex 'target remote localhost:1234'
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 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-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...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000000100 in ?? ()
(gdb) x/10i 0x80200000
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) █
```

阶段三：设置断点并观察控制权转移

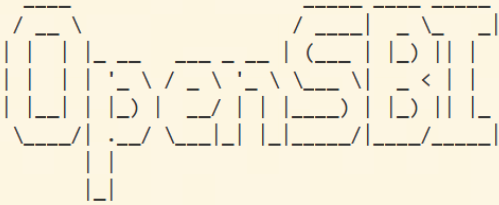
我在内核的入口处用指令 `b kern_entry` 设置了一个断点，一旦cpu运行到这里就会停止。接下来开始运行，发现程序成功停下来了。

便于观察，我使用 `disassemble` 查看当前位置附近的汇编代码，看到 `=>` 指向即将被执行的 `la sp, bootstacktop` 指令。

然后输入 `si` 单步执行这条指令，再通过 `disassemble` 来观察，发现跳转到下一条，证明我们的运行顺利进行了。


```
yqqos@yxq:~/labcode/lab1$ make debug
```

OpenSBI v0.9



```
Platform Name      : riscv-virtio,qemu
Platform Features  : timer,mfdeleg
Platform HART Count : 1
Firmware Base      : 0x80000000
Firmware Size      : 100 KB
Runtime SBI Version : 0.2

Domain0 Name       : root
Domain0 Boot HART  : 0
Domain0 HARTs      : 0*
Domain0 Region00   : 0x0000000080000000-0x000000008001ffff
()
Domain0 Region01   : 0x0000000000000000-0xffffffff
ffff (R,W,X)
Domain0 Next Address : 0x0000000080200000
Domain0 Next Arg1   : 0x0000000087000000
Domain0 Next Mode    : S-mode
Domain0 SysReset    : yes

Boot HART ID       : 0
Boot HART Domain    : root
Boot HART ISA       : rv64imafdcsu
Boot HART Features  : scounteren,mcounteren,time
Boot HART PMP Count : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG   : 0x0000000000000222
Boot HART MEDELEG   : 0x000000000000b109
```

```
0x0000000000001000 in ?? ()
(gdb) x/10i 0x80200000
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) b kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) c
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) disassemble
Dump of assembler code for function kern_entry:
=> 0x0000000080200000 <+0>: auipc sp,0x3
0x0000000080200004 <+4>: mv sp,sp
0x0000000080200008 <+8>: j 0x8020000a <kern_in
it>
End of assembler dump.
(gdb) si
0x0000000080200004 in kern_entry ()
at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) disassemble
Dump of assembler code for function kern_entry:
=> 0x0000000080200000 <+0>: auipc sp,0x3
0x0000000080200004 <+4>: mv sp,sp
0x0000000080200008 <+8>: j 0x8020000a <kern_in
it>
End of assembler dump.
(gdb) i r sp
sp      0x80203000      0x80203000 <SBI_CONSOLE_PUT
CHAR>
(gdb) si
```

我又设置了一个断点，观察到了从汇编到C语言环境的跳转。

```

(gdb) b kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
(gdb) c
Continuing.

Breakpoint 2, kern_init () at kern/init/init.c:8
8      memset(edata, 0, end - edata);
(gdb) disassemble
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>:      auipc    a0,0x3
      0x000000008020000e <+4>:      addi     a0,a0,-2 # 0x80203008
      0x0000000080200012 <+8>:      auipc    a2,0x3
      0x0000000080200016 <+12>:     addi     a2,a2,-10 # 0x80203008
      0x000000008020001a <+16>:     addi     sp,sp,-16
      0x000000008020001c <+18>:     li       a1,0
      0x000000008020001e <+20>:     sub      a2,a2,a0
      0x0000000080200020 <+22>:     sd       ra,8(sp)
      0x0000000080200022 <+24>:     jal      ra,0x802004b6 <memset>
      0x0000000080200026 <+28>:     auipc    a1,0x0
      0x000000008020002a <+32>:     addi     a1,a1,1186 # 0x802004c8
      0x000000008020002e <+36>:     auipc    a0,0x0
      0x0000000080200032 <+40>:     addi     a0,a0,1210 # 0x802004e8
      0x0000000080200036 <+44>:     jal      ra,0x80200056 <cprintf>
      0x000000008020003a <+48>:     j        0x8020003a <kern_init+48>

End of assembler dump.
(gdb)

```

在kern_init函数中，我使用n (next)命令逐行执行C代码。当执行完cprintf(...)函数后，我立刻在左侧的QEMU窗口中观察到了 (THU.CST) os is loading ...的输出。这一现象完美地将GDB调试器中的指令执行与操作系统的实际外部表现联系了起来，验证了我们自己构建的打印函数栈（cprintf -> cons_putc -> sbi_call -> ecall）是正确且有效的。

```

Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000080700000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA              : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
(THU.CST) os is loading ...

(gdb) disassemble
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>:      auipc    a0,0x3
      0x000000008020000e <+4>:      addi     a0,a0,-2 # 0x80203008
      0x0000000080200012 <+8>:      auipc    a2,0x3
      0x0000000080200016 <+12>:     addi     a2,a2,-10 # 0x80203008
      0x000000008020001a <+16>:     addi     sp,sp,-16
      0x000000008020001c <+18>:     li       a1,0
      0x000000008020001e <+20>:     sub      a2,a2,a0
      0x0000000080200020 <+22>:     sd       ra,8(sp)
      0x0000000080200022 <+24>:     jal      ra,0x802004b6 <memset>
      0x0000000080200026 <+28>:     auipc    a1,0x0
      0x000000008020002a <+32>:     addi     a1,a1,1186 # 0x802004c8
      0x000000008020002e <+36>:     auipc    a0,0x0
      0x0000000080200032 <+40>:     addi     a0,a0,1210 # 0x802004e8
      0x0000000080200036 <+44>:     jal      ra,0x80200056 <cprintf>
      0x000000008020003a <+48>:     j        0x8020003a <kern_init+48>

End of assembler dump.
(gdb) n
11      cprintf("%s\n", message);
(gdb) n
12      while (1)
(gdb)

```

问题1：RISC-V 硬件加电后最初执行的几条指令位于什么地址？

答案：通过GDB调试器在连接QEMU后直接观察程序计数器（`pc`）的初始值，可以确定，在加电复位后，最初执行的指令位于物理地址 `0x1000`。

问题2：它们主要完成了哪些功能？

答案：位于 `0x1000` 处的指令是固化在模拟ROM中的**复位向量代码**，它并不属于OpenSBI固件。核心功能为：

- 1. **最简硬件初始化**：进行CPU启动所必需的最基本的状态设置，例如配置机器模式（M-Mode）下的异常/中断入口地址，即将一个预设的地址写入 `mtvec` 寄存器。
- 2. **引导加载与控制权移交**：这段代码的最终使命是执行一条无条件的跳转指令，将CPU的控制权移交给一个引导固件**OpenSBI**。

重要知识点

特权级与特权指令

在计算机中，**固件(firmware)** 是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。固件可以为设备更复杂的软件（如操作系统）提供标准化的操作环境。对于不太复杂的设备，固件可以直接充当设备的完整操作系统，执行所有控制、监视和数据操作功能。在基于 x86 的计算机系统中, BIOS 或 UEFI 是固件；在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在**M态（M-mode）**，因为固件需要直接访问硬件。

RISCV有四种**特权级（privilege level）**。

| Level | Encoding | 全称 | 简称 |
|-------|----------|--------------------|----|
| 0 | 00 | User/Application | U |
| 1 | 01 | Supervisor | S |
| 2 | 10 | Reserved(目前未使用，保留) | |
| 3 | 11 | Machine | M |

`ecall` 指令是 RISC-V 中用于实现受控的权限提升的关键指令。

当在 U 模式（用户态）执行 `ecall`，会触发异常，从而陷入到 S 模式（内核态）。这是系统调用的底层机制。

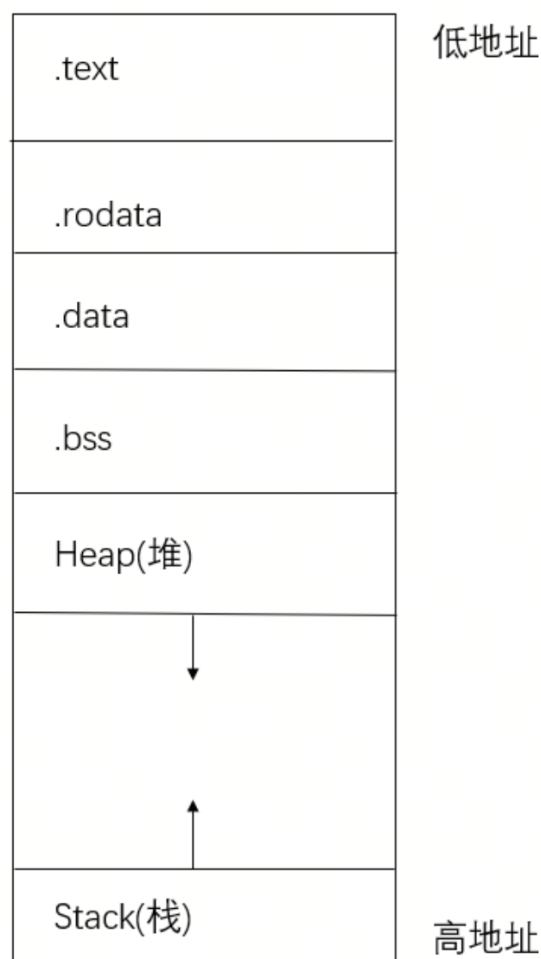
当在 S 模式（内核态）执行 `ecall`，会触发异常，从而陷入到 M 模式（机器态）。这正是我们调用 `OpenSBI` 服务的方式。

两种不同的可执行文件格式

`elf` 文件([wikipedia: elf](https://en.wikipedia.org/wiki/elf_format))比较复杂, 包含一个文件头(ELF header), 包含冗余的调试信息, 指定程序每个section的内存布局, 需要解析program header才能知道各段(section)的信息。如果我们已经有一个完整的操作系统来解析elf文件, 那么elf文件可以直接执行。但是对于OpenSBI来说, elf格式还是太复杂了。

`bin` 文件就比较简单了, 简单地在文件头之后解释自己应该被加载到什么起始位置。

elf文件的内存布局



- `.text` 段, 即代码段, 存放汇编代码;
- `.rodata` 段, 即只读数据段, 顾名思义里面存放只读数据, 通常是程序中的常量;
- `.data` 段, 存放被初始化的可读写数据, 通常保存程序中的全局变量;
- `.bss` 段, 存放被初始化为 00 的可读写数据, 与 `.data` 段的不同之处在于我们知道它要被初始化为 00, 因此在可执行文件中只需记录这个段的大小以及所在位置即可, 而不用记录里面的数据。
- `stack`, 即栈, 用来存储程序运行过程中的局部变量, 以及负责函数调用时的各种机制。它从高地址向低地址增长;
- `heap`, 即堆, 用来支持程序运行过程中内存的动态分配, 比如说你要读进来一个字符串, 在你写程序的时候你也不知道它的长度究竟为多少, 于是你只能在运行过程中, 知道了字符串的长度之后, 再在堆中给这个字符串分配内存。

make 和 Makefile

`Makefile` 文件是一个**自动化构建工具** `make` 的配置文件。核心作用是**定义和管理源代码的编译规则和文件依赖关系**，从而实现自动化、高效的构建流程。

makefile的基本规则

在使用这个makefile之前，还是让我们先来粗略地看一看makefile的规则。

代码块

```
1  target ... : prerequisites ...  
2      command  
3      ...  
4      ...
```

target也就是一个目标文件，可以是object file，也可以是执行文件。还可以是一个标签（label）。prerequisites就是，要生成那个target所需要的文件或是目标。command也就是make需要执行的命令（任意的shell命令）。这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在 command中。如果prerequisites中有一个以上的文件比target文件要新，那么command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容