

练习1：理解内核启动中的程序入口操作

操作系统内核启动流程

首先，要了解 `kern/init/entry.s` 内容代码，我们需要先复习下操作系统内核启动流程的相关知识：

操作系统启动时，会经历下面几个阶段：

- 由BIOS/UEFI完成硬件初始化
- 加载引导加载程序（Bootloader）
- Bootloader将内核镜像加载到内存中
- Bootloader将控制权交给内核入口点。

内核启动后，入口点代码（如 `kern_entry`）会完成栈指针的初始化，为后续函数调用提供支持。随后，内核跳转到初始化函数（如 `kern_init`），完成页表设置、中断初始化、设备驱动加载等操作，最终启动第一个用户态进程，进入正常运行状态。

代码解释

`kern/init/entry.s` 中，在下方有设置栈帧的一部分代码

```
1  .section .data
2      # .align 2^12
3      .align PGSHIFT
4      .global bootstack
5  bootstack:
6      .space KSTACKSIZE
7      .global bootstacktop
8  bootstacktop:
```

问题回答

然后，作业要求中要解释两行代码：

```
1  | 1a sp, bootstacktop
```

这行代码完成了设置栈帧，`bootstacktop`是刚才设置的栈帧的栈顶，我们在这行代码中将指向栈顶指针的寄存器`sp`存上栈顶`bootstacktop`。完成这一步，我们就为后续的函数调用做好了准备。

```
1  | tail kern_init
```

接下来，这是一行无返回跳转指令，采用尾调用优化，表示直接内核将控制权转移给内核初始化函数 `kern_init`，随后，这个函数将开始执行内核的主要逻辑，并完成以下任务：

1. 设置页表和虚拟内存管理
2. 初始化中断向量和中断处理机制
3. 初始化设备驱动程序
4. 启动调度器，加载第一个用户态进程

GDB调试信息

我们可以在 GDB 中进行以下调试：

```
(gdb) info address bootstack
Symbol "bootstack" is at 0x80201000 in a file compiled without debugging.
(gdb) info address bootstacktop
Symbol "bootstacktop" is at 0x80203000 in a file compiled without debugging.
(gdb) info registers sp
sp          0x0          0x0
(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) info address bootstack
Symbol "bootstack" is at 0x80201000 in a file compiled without debugging.
(gdb) info address bootstacktop
Symbol "bootstacktop" is at 0x80203000 in a file compiled without debugging.
(gdb) info registers sp
sp          0x8001bd80      0x8001bd80
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) info registers sp
sp          0x80203000      0x80203000 <SBI_CONSOLE_PUTCHAR>
(gdb) █
```

如上所示，我们查看符号地址：

```
1 | (gdb) info address bootstack
2 | (gdb) info address bootstacktop
```

输出

```
1 | Symbol "bootstack" is at 0x80201000
2 | Symbol "bootstacktop" is at 0x80203000
```

查看设置栈前后 sp 的变化：

执行指令

```
1 | (gdb) info registers sp
```

我们发现，一开始，

```
1 | sp = 0x0
```

我们在 `kern_entry` 处设置断点之后执行程序在此处中断，然后再单步执行之后查看 `sp` 寄存器，执行指令：

```
1 | (gdb) si
2 | (gdb) info registers sp
```

我们可以发现 `sp` 的值被设置为了 `0x80203000`，这与 `bookstacktop` 一致，说明栈帧已经成功创建！

继续执行并观察跳转：

```
1 (gdb) si
2 (gdb) info registers pc
```

此时 `pc` 跳到了 `kern_init` 的地址，说明 `tail kern_init` 已经执行。

练习2: 使用GDB验证启动流程

首先，我们使用 `make` 命令来测试代码的 `makefile`，得到结果如下：

```
(base) lamp@gsags:/mnt/d/desktop/os/lab1$ make
make: Warning: File 'obj/kern/driver/console.d' has modification time 1.8 s in the future
+ cc kern/init/entry.S
+ cc kern/init/init.c
+ cc kern/libs/stdio.c
+ cc kern/driver/console.c
+ cc libs/printfmt.c
+ cc libs/readline.c
+ cc libs/sbi.c
+ cc libs/string.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img
make: warning: Clock skew detected. Your build may be incomplete.
```

`make`成功，说明我们的 `makefile` 是正确的。

接着，我们使用 `makefile` 里定义好的命令行，利用 `make` 后生成的 `ucore.img` 打开 `qemu` 模拟器：

```
1 make debug
```

这样，我们的 `qemu` 就已经启动并在后台运行了。接下来，我们再开启一个 `ws` 窗口，运行 `make gdb` 并连接到 `qemu`：

运行结果如下：

```
(base) lamp@gsags:/mnt/d/desktop/os/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
0x00000000000001000 in ?? ()
```

然后，我们使用 `info registers` 查看所有寄存器的信息：

```
(gdb) info registers
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
s9          0x0      0
s10         0x0      0
s11         0x0      0
t3          0x0      0
t4          0x0      0
t5          0x0      0
t6          0x0      0
pc          0x1000   0x1000
(gdb) █
```

发现输出了寄存器的信息，说明我们连接成功了。

然后我们使用 `x/20i $pc` 命令查看当前位置的最近20条指令，输出如下：

```
(gdb) x/20i $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:      0x8000
    0x101c:      unimp
    0x101e:      unimp
    0x1020:      unimp
    0x1022:      0x8700
    0x1024:      unimp
    0x1026:      unimp
    0x1028:      fnmadd.s      ft6,ft4,fs4,fs1,unknown
    0x102c:      unimp
    0x102e:      unimp
    0x1030:      c.slli64      zero
    0x1032:      unimp
    0x1034:      unimp
```

我们可以看到，0x1014 上的jr指令跳转到了 t0 寄存器的位置，这里应该是重点，因为之后的命令都不会再被执行

我们使用 si 进行单步执行，然后查看涉及到的寄存器的值，来分析发生了什么变化：

```
1 (gdb) si
2 (gdb) info r t0
3 t0          0x1000    4096
4 (gdb) si
5 0x00000000000001008 in ?? ()
6 (gdb) info r a2
7 a2          0x1028    4136
8 (gdb) si
9 0x0000000000000100c in ?? ()
10 (gdb) info r a0
11 a0          0x0        0
12 (gdb) si
13 0x00000000000001010 in ?? ()
14 (gdb) info r a1
15 a1          0x87000000    2264924160
16 (gdb) si
17 0x00000000000001014 in ?? ()
18 (gdb) info r t0
19 t0          0x80000000    2147483648
```

分析以上指令和相关寄存器的值的变化，我们可以看到，程序先将当前PC地址(0x1000)加载到t0寄存器，作为基地址（因此第一次查看t0的值变成了0x1000），然后让a2指向t0+40（因此a2变为0x1028），再让a0读取硬件线程id，再从t0+32加载数据到a1。

以上均不是重点，在第六条指令（0x1014 的 jr t0）上，我们从地址 0x1018（t0+24）上加载数据存到t0作为跳转目标。可以看到这次指令执行完毕之后，t0的值变为了 0x80000000，说明我们将要跳转到 0x80000000 的位置。

我们继续执行 `si`，执行这步 `jr`，发现跳转到了 `0x80000000`

```
1 (gdb) si
2 0x0000000080000000 in ?? ()
```

此时我们再查看下附近的20条指令

```
(gdb) x/20i $pc
=> 0x80000000: add    s0,a0,zero
    0x80000004: add    s1,a1,zero
    0x80000008: add    s2,a2,zero
    0x8000000c: jal    ra,0x800006a0
    0x80000010: add    a6,a0,zero
    0x80000014: add    a0,s0,zero
    0x80000018: add    a1,s1,zero
    0x8000001c: add    a2,s2,zero
    0x80000020: li     a7,-1
    0x80000022: beq    a6,a7,0x8000002a
    0x80000026: bne    a0,a6,0x80000160
    0x8000002a: auipc  a6,0x12
    0x8000002e: ld     a6,1318(a6)
    0x80000032: li     a7,1
    0x80000034: amoadw a6,a7,(a6)
    0x80000038: bnez   a6,0x80000160
    0x8000003c: auipc  t0,0x12
    0x80000040: ld     t0,1308(t0)
    0x80000044: auipc  t1,0x12
    0x80000048: ld     t1,1444(t1)
```

由于这部分代码很长，因此我们直接设置断点然后观察SBI何时将我们内核代码写入 `0x80200000`

```
1 watch *kern_entry
2 c
```

在这之后，终端卡死在 `continue` 界面没有输出，我们手动 `ctrl+c` 中断debug，发现程序停止在 `kern_init` 里。这是因为，`watch` 指令是检测到观测点的数值变化才会停止，但是我们的代码一开始就将内核加载，所以 `0x80200000` 不会有数值变化。

```

(base) lamp@gsags:/mnt/d/desktop/os/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) watch *0x80200000
Hardware watchpoint 1: *0x80200000
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
kern_init () at kern/init/init.c:12
12      while (1)
(gdb) █

```

接下来，我们重新打开两个窗口，进入qemu和gdb，然后这次，我们使用

```

1 | b *kern_entry
2 | c

```

目的是进入 `kernel.ld` 文件中所定义的 `base_address`，也就是加载地址的位置。在执行完这个命令之后，得到的输出结果如下：

```
(base) lamp@gsags:/mnt/d/desktop/os/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
0x0000000000001000 in ?? ()
(gdb) b *0x80200000
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)
```

可以看到，我们停在了entry.s里的 `la sp bootstacktop` 这一行代码中，说明在这里，我们加载地址执行的汇编代码是这一行，也就是给栈顶指针sp赋值的这一块，同时，他也是我们定义的 `kern_entry` 这个标签下的代码位置。

接下来，我们再使用 `x/10i $pc` 观察下 `0x80200000` 附近的十条指令：

```
1  => 0x80200000 <kern_entry>:      auipc    sp,0x3
2      0x80200004 <kern_entry+4>:    mv        sp,sp
3      0x80200008 <kern_entry+8>:    j         0x8020000a <kern_init>
4      0x8020000a <kern_init>:      auipc    a0,0x3
5      0x8020000e <kern_init+4>:    addi     a0,a0,-2
6      0x80200012 <kern_init+8>:    auipc    a2,0x3
7      0x80200016 <kern_init+12>:   addi     a2,a2,-10
8      0x8020001a <kern_init+16>:   addi     sp,sp,-16
9      0x8020001c <kern_init+18>:   li       a1,0
10     0x8020001e <kern_init+20>:   sub      a2,a2,a0
```

我们查看下栈顶指针附近的十个内存单元的内容：

```
1  (gdb) x/10x $sp
2  0x8001bd80:    0x8001be00    0x00000000    0x8001be00    0x00000000
3  0x8001bd90:    0x46444341    0x55534d49    0x00000000    0x00000000
4  0x8001bda0:    0x00000000    0x00000000
```

可以发现，是一系列杂乱的数据。

接下来我们单步执行，然后再次查看sp附近十个单元的内容：


```

1  gdb) si
2  0x0000000080200004 in kern_entry () at kern/init/entry.S:7
3  7          la sp, bootstacktop
4  (gdb) x/10x $sp
5  0x80203000 <SBI_CONSOLE_PUTCHAR>:      0x00000001      0x00000000
6  0x00000000x00000000
7  0x80203010:      0x00000000      0x00000000      0x00000000      0x00000000
8  0x80203020:      0x00000000      0x00000000

```

发现已经基本都被清零，说明已经完成了内存清零的工作。

我们再单步执行几次，发现这一段内存全都是清零的了。

```

1  (gdb) si
2  9          tail kern_init
3  (gdb) x/10x $sp
4  0x80203000 <SBI_CONSOLE_PUTCHAR>:      0x00000001      0x00000000
5  0x00000000x00000000
6  0x80203010:      0x00000000      0x00000000      0x00000000      0x00000000
7  0x80203020:      0x00000000      0x00000000
8  (gdb) si
9  kern_init () at kern/init/init.c:8
10 8          memset(edata, 0, end - edata);
11 (gdb) x/10x $sp
12 0x80203000 <SBI_CONSOLE_PUTCHAR>:      0x00000001      0x00000000
13 0x00000000x00000000
14 0x80203010:      0x00000000      0x00000000      0x00000000      0x00000000
15 0x80203020:      0x00000000      0x00000000
16 (gdb)

```

在这一段过程中，曾有过疑问：内存清零不是kerninit里才干的事情吗？为什么在kernentry里单步执行，还没tail kern init就已经清零了？

在询问大模型以及自己思考之后，可以给出回答：kernentry这行汇编代码指令的作用是把bootstacktop赋值给sp，于是我们要查看的sp附近的内存单元就跳转到了botstacktop这里。所以，原来显示的是没有被清零的内存数据，执行了这条指令之后就变成了已经被清空过的bss段的数据。这是展示位置发生了变化。通过观察我们先后查看寄存器的地址，我们也可以看见，刚到kern_entry的时候，查看的地址是0x8001bd80，0x8001bd90，0x8001bda0，而单步执行后，看到的地址全都是0x80203000，0x80203010，0x80203020等，

在kern_entry设置了栈顶之后，就进入了我们用c语言自定义的内核初始化函数kern_init。然后我们追踪到kern_init，打好断点，

然后一路执行过去：

```

(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);

```

可以看到，我们进入了init.c文件中定义的kern_init()函数的第二行代码中。这行代码的意思是将栈空间重置为0。至于为什么不是跳转到第一行的 `extern char edata[], end[]`；是因为，它是变量的声明，而不是实际的代码执行。它告诉编译器，这两个符号在其他地方定义（通常是链接脚本），但不会生成任何指令。

然后我们可以查看一下 `kern_init()` 函数的完整反汇编代码（使用 `disassemble kern_init` 指令）：

```
1 (gdb) disassemble kern_init
2 Dump of assembler code for function kern_init:
3 => 0x000000008020000a <+0>:      auipc    a0,0x3
4      0x000000008020000e <+4>:      addi     a0,a0,-2 # 0x80203008
5      0x0000000080200012 <+8>:      auipc    a2,0x3
6      0x0000000080200016 <+12>:     addi     a2,a2,-10 # 0x80203008
7      0x000000008020001a <+16>:     addi     sp,sp,-16
8      0x000000008020001c <+18>:     li       a1,0
9      0x000000008020001e <+20>:     sub     a2,a2,a0
10     0x0000000080200020 <+22>:     sd      ra,8(sp)
11     0x0000000080200022 <+24>:     jal     ra,0x802004b6 <memset>
12     0x0000000080200026 <+28>:     auipc    a1,0x0
13     0x000000008020002a <+32>:     addi     a1,a1,1186 # 0x802004c8
14     0x000000008020002e <+36>:     auipc    a0,0x0
15     0x0000000080200032 <+40>:     addi     a0,a0,1210 # 0x802004e8
16     0x0000000080200036 <+44>:     jal     ra,0x80200056 <cprintf>
17     0x000000008020003a <+48>:     j       0x8020003a <kern_init+48>
18 End of assembler dump.
```

这里针对之前没有学过的新指令 `auipc` 给出一些笔记：它的作用是将立即数加到当前 PC 的高位，也就是将当前程序计数器（PC）的高 20 位加上一个立即数（`imm`），结果存入目标寄存器。

例如，第一行代码 `auipc a0,0x3` 的意思就是 $a0 = pc + 0x3 \ll 12$ ，这行指令的目的是计算出全局变量的基地址，然后让下一行指令再对低位进行细微的调整来获得最终的变量地址。

在前四行，我们分别将 `edata`、`end` 的地址计算出并保存到 `a0` 和 `a2` 寄存器中，然后保存栈顶指针到 `ra` 寄存器里，在 `0x0000000080200022` 使用 `jal` 跳转调用 `memset` 函数进行清零操作。

值得注意的是，我们有一个 `while(1)` 的死循环代码，在这里，就是使用 `j` 指令跳转到当前 `j` 指令的地址，形成一个死循环。

这是显然的，因为从调试信息我们呢可以看到，`j` 指令的地址就是 `kern_init+48`，跟我们 `j` 指令跳转的目的地址一样。

然后再输入 `c` 进行 `continue`，发现 `qemu` 的 `debug` 窗口出现了我们在 `kern_init` 函数中定义的输出：`(THU.CST) os is loading ... \n`，也代表我们的代码运行完毕了。具体执行效果如下：

```
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
```

问题回答

问题 1: RISC-V 硬件加电后最初执行的指令位于什么地址?

问题 2: 这些指令主要完成了哪些功能?

随后，SBI固件会进行

1. 初始化中断控制器用于管理定时器中断和外部中断
2. 配置物理内存保护，确保内核只能访问合法的内存区域
3. 配置硬件计时器，为内核提供时间管理功能
4. 将内核镜像从存储设备（如磁盘或 ROM）加载到内存中的指定地址
5. 将控制权交给内核的入口地址（如 `0x80200000`），开始执行内核代码。

等一系列功能。