

Lab1实验报告

小组成员：2311858 许哲涵，2311576 赵宁，2313650 赵悦蛟

一.理解内核启动中的程序入口操作

1. 指令 `la sp, bootstacktop` 操作及目的

`la` 是汇编中的 "加载地址" 指令，`sp` 是栈指针寄存器 (Stack Pointer)。这条指令的作用是将 `bootstacktop` 符号所代表的内存地址加载到栈指针寄存器 `sp` 中，即设置系统栈的栈顶地址。

代码中 `.space KSTACKSIZE` 定义了一块大小为 `KSTACKSIZE` 的内存区域作为内核启动栈 (`bootstack`)，而 `bootstacktop` 是该栈的顶部地址 (栈在内存中从高地址向低地址增长，因此栈顶是栈空间的起始高地址)。

目的：内核启动初期 (尚未建立完整的内存管理系统时)，必须先初始化一个临时栈。栈是函数调用、局部变量存储、中断处理等操作的基础——没有栈，后续的函数调用 (如 `kern_init`) 无法正常执行。这条指令的核心作用是为内核启动阶段提供一个可用的栈空间，确保后续代码能正确运行。

2. `tail kern_init` 操作及目的

`tail` 是一种特殊的跳转指令 (类似于 `jmp`，但会优化函数调用的栈帧处理)，作用是跳转到 `kern_init` 函数执行，且不会在栈上保存返回地址。`kern_init` 是内核初始化的 C 语言入口函数 (这段汇编是内核启动的最早期代码，之后会移交控制权给 C 语言实现的初始化逻辑)。

目的：内核启动流程分为汇编阶段和 C 语言阶段。汇编阶段的任务极其简单 (仅初始化栈等最基础的环境)，完成后需要将控制权转交给更复杂的 C 语言初始化逻辑 (如初始化内存管理、中断系统、进程调度等)。使用 `tail` 而非普通函数调用 (如 `call`)，是因为内核启动后不会再返回这个汇编入口，因此无需保存返回地址，这样可以优化栈空间的使用。其核心目的是完成从汇编启动代码到 C 语言内核初始化函数的过渡。

二. 使用GDB验证启动流程

我们需要编译所有的源代码，用目标文件链接起来，生成 `elf` 文件，生成 `bin` 硬盘镜像，用 `qemu` 跑起来。在使用 `make` 命令执行时，需要 `makefile` 文件，以告诉 `make` 命令需要怎么样的去编译和链接程序。我们在源代码的目录下 `make qemu`，以启动 `qemu`，得到如下运行结果，说明我们已经把 `ucore` 跑起来了。



bin



kern



libs



obj



tools



Makefile

```
ubuntu@ubuntu-VMware-Virtual-Platform: ~/labcode/lab1
ubuntu@ubuntu-VMware-Virtual-Platform:~/labcode/lab1$ make qemu
+ 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

OpenSBI v0.4 (Jul  2 2019 11:53:53)

      _ _ _ _ _      _ _ _ _ _      _ _ _ _ _
    /  _  \      /  _ _ \  _  \  _  \
 |  |  |  | _ _ _ _ _ _ _ _ _ | ( _ _ |  |  | | | |
 |  |  |  | ' _ \ / _ \ ' _ \ \ _ \ |  < |  |
 |  | _ |  | ) |  _/ |  | ( _ _ ) |  |  |  |
 \ _ _ / | . _ / \ _ _ |  |  | _ _ / |  _ / _ _ |
    |  |
    | _ |

Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
```

有了上述运行结果，说明我们的 QEMU 已经跑起来了。但是，我们还需要命令行调试工具——GDB 来进行调试。我们让 QEMU 扮演被调试的目标，它按照我们的要求启动内核，然后在某个端口上等待；同时，启动 GDB 去连接 QEMU 等待的那个端口。这样 GDB 就能向我们报告 QEMU 内部虚拟 CPU 的一举一动，我们就能调试这个内核了。

我们打开了两个终端窗口，一个输入 `make debug`，这个命令会启动 QEMU；另一个窗口我们输入 `make gdb`，这个命令包含一些列操作，其中 `file bin/kernel` 让 GDB 加载我们编译好的内核文件，`set arch riscv:rv64` 则告诉 GDB，我们要调试的是 RISC-V 64 位的程序，`target remote localhost:1234` 则让 GDB 去连接本机（localhost）的 1234 端口，也就是 QEMU 正在等待我们的地方。

```
ubuntu@ubuntu-VMware-Virtual-Platform: ~/labcode/lab1
ubuntu@ubuntu-VMware-Virtual-Platform:~/labcode/lab1$ make debug

ubuntu@ubuntu-VMware-Virtual-Platform:~/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 8.3.0-2020.04.1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://www.gnu.org/licenses/gpl.html>
```

当 GDB 成功连上 QEMU 后，它会告诉我们：“现在程序停在了地址 0x1000 这个地方”。接下来我们输入指令 `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，所以实际只会执行到 0x1010 的指令处。

```
ubuntu@ubuntu-VMware-Virtual-Platform: ~/labcode/lab1

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 assumed to be riscv:rv64
Remote debugging using localhost:1234
0x00000000000001000 in ?? ()
(gdb) x/10i $pc
=> 0x1000:      auipc    t0,0x0
0x1004:      addi     a1,t0,32
0x1008:      csrr     a0,mhartid
0x100c:      ld       t0,24(t0)
0x1010:      jr       t0
0x1014:      unimp
0x1016:      unimp
0x1018:      unimp
0x101a:      0x8000
0x101c:      unimp
```

接下来解释一下这段汇编程序

```

0x1000: auipc t0,0x0      #将当前程序计数器 (PC) 的值加上立即数左移 12 位的结果, 存入寄存器 t0
0x1004: addi a1,t0,32     #将寄存器 t0 的值加上立即数 32, 结果存入 a1 寄存器
0x1008: csrr a0,mhartid  #读取控制状态寄存器的值到通用寄存器 a0, a0 = mhartid = 0
0x100c: ld t0,24(t0)     #从内存地址 (t0 + 24) 处加载 64 位数据到 t0 寄存器, t0 = [t0 + 24] =
0x80000000, 0x80000000 是预先安排好的下一阶段执行入口
0x1010: jr t0           #跳转到 t0 寄存器中存储的地址, 也就是0x80000000

```

接下来, 我们输入 `si` 进行单步执行, 使用如 `info r t0` 查看每一步执行完涉及到的寄存器结果。我们也可以发现执行到 `0x1010` 时, `t0` 寄存器的值改变了。

```

(gdb) si
0x00000000000001004 in ?? ()
(gdb) info r t0
t0                0x1000    4096
(gdb) si
0x00000000000001008 in ?? ()
(gdb) info r t0
t0                0x1000    4096
(gdb) si
0x0000000000000100c in ?? ()
(gdb) info r t0
t0                0x1000    4096
(gdb) si
0x00000000000001010 in ?? ()
(gdb) info r t0
t0                0x80000000    2147483648
(gdb) si
0x0000000000000000 in ?? ()

```

之后会跳转到 `0x80000000` 处继续执行, 我们输入 `x/10i 0x80000000`, 可以显示 `0x80000000` 处的10条指令如下:

```

(gdb) x/10i 0x80000000
=> 0x80000000: csrr    a6,mhartid
    0x80000004: bgtz    a6,0x80000108
    0x80000008: auipc   t0,0x0
    0x8000000c: addi    t0,t0,1032
    0x80000010: auipc   t1,0x0
    0x80000014: addi    t1,t1,-16
    0x80000018: sd      t1,0(t0)
    0x8000001c: auipc   t0,0x0
    0x80000020: addi    t0,t0,1020
    0x80000024: ld      t0,0(t0)

```

接下来我们逐条解释一下：

```
0x80000000: csrr a6,mhartid    #读取当前硬件线程(hart)的ID到a6寄存器
0x80000004: bgtz a6,0x80000108 #如果a6的值大于0,则跳转到0x80000108处执行
0x80000008: auipc t0,0x0    #将当前PC值(0x80000008)加载到t0寄存器
0x8000000c: addi t0,t0,1032    #t0 = 0x80000008 + 1032 = 0x80000408
0x80000010: auipc t1,0x0    #将当前PC值(0x80000010)加载到t1寄存器
0x80000014: addi t1,t1,-16    #t1 = 0x80000010 - 16 = 0x80000000
0x80000018: sd t1,0(t0)    #将t1的值(0x80000000)存储到内存地址t0+0(即0x80000408)
0x8000001c: auipc t0,0x0    #将当前PC值(0x8000001c)加载到t0寄存器
0x80000020: addi t0,t0,1020    #t0 = 0x8000001c + 1020 = 0x80000400
0x80000024: ld t0,0(t0)    #从内存地址0x80000400处加载一个64位数据到t0寄存器
```

这段代码用来加载操作系统并启动操作系统。

接下来输入指令 `break kern_entry`，`kern_entry` 是要设置断点的目标位置，输出如下所示：

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

其中 `Breakpoint 1` 表示这是设置的第 1 个断点。 `at 0x80200000`：说明断点设置在内存地址 `0x80200000` 处。
`file kern/init/entry.S, line 7` 指出该断点对应的源代码文件是 `kern/init/entry.S`，位于这个文件的第 7 行。这意味着当程序执行到 `kern/init/entry.S` 文件的第 7 行（或者说程序运行到内存地址 `0x80200000` 处的指令）时，`GDB` 会暂停程序的执行，方便进行调试操作，比如查看寄存器值、内存内容、变量值等。

接着输入指令 `x/5i 0x80200000`，查看汇编代码如下所示：

```
(gdb) x/5i 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
```

我们逐一进行解释：

```
0x80200000 <kern_entry>: auipc sp,0x3 #sp = 当前PC值(0x80200000) + (0x3 << 12) =
0x80200000 + 0x3000 = 0x80203000, 为内核执行准备栈空间
0x80200004 <kern_entry+4>: mv sp,sp #将源寄存器的值复制到目标寄存器, 实际就是把自己的值赋给自己
0x80200008 <kern_entry+8>: j 0x8020000c <kern_init> #从kern_entry跳转到kern_init函数继续执行
0x8020000a <kern_init>: auipc a0,0x3 #a0 = PC + (0x3 << 12)=0x8020300a
0x8020000e <kern_init+4>: addi a0,a0,-2 #a0 = 0x8020300a - 2 = 0x80203008
```

这段代码主要是为内核的后续初始化搭建基础运行环境，在 `kern_entry` 之后通过 `j` 指令跳转到 `kern_init` 函数，进入内核初始化流程，在 `kern_init` 中，又通过 `auipc` 和 `addi` 指令计算并调整参数（存在 `a0` 中），为后续初始化操作传递必要的信息。接下来我们输入 `continue`，一直执行到断点，再进行 `debug` 可以得到如下输出：

[illegible]

说明此时 `OpenSBI` 已经启动，接着我们输入指令 `break kern_init`，输出如下：

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

发现就是指向了之前显示为 `<kern_init>` 的地址 `0x8020000a`。再次输入 `continue`，再输入 `disassemble kern_init` 查看反汇编代码，结果如下：

The image displays two terminal windows side-by-side, both running on a system identified as 'ubuntu@ubuntu-VMware-Virtual-Platform: ~/labcode/la'.

The left terminal window shows the OpenSBI v0.4 boot log, dated July 2, 2019, at 11:53:53. It features a stylized ASCII art logo of a computer monitor. Below the logo, the following platform information is displayed:

```
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
```

At the bottom of the left window, the following messages are visible:

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

The right terminal window shows the GDB disassembly of the `kern_init` function. The breakpoint is set at `kern/init/init.c:8`. The disassembly code is as follows:

```
8      memset(edata, 0, end - edata);
(gdb) disassemble kern_init
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,0x802004be <memset>
      0x0000000080200026 <+28>:     auipc    a1,0x0
      0x000000008020002a <+32>:     addi     a1,a1,1194 # 0x802004d0
      0x000000008020002e <+36>:     auipc    a0,0x0
      0x0000000080200032 <+40>:     addi     a0,a0,1218 # 0x802004f0
      0x0000000080200036 <+44>:     jal     ra,0x80200056 <cprintf>
      0x000000008020003a <+48>:     j       0x8020003a <kern_init+48>
```

Below the disassembly, the following messages are shown:

```
End of assembler dump.
(gdb) continue
Continuing.
```

观察结果，我们可以发现最后一个指令是 `j 0x8020003a <kern_init+48>`，这条指令的作用是：无条件跳转到 `kern_init` 函数内部偏移 48 字节的位置（具体地址为 `0x8020003a`）继续执行。同时，如果在查看完反汇编代码后继续输入 `continue`，我们可以在 debug 窗口里看到 (THU.CST) `os is loading ...` 的输出。

三.思考与回答

1. RISC-V 硬件加电后最初执行的几条指令位于什么地址？

RISC-V 硬件加电后，最初执行的几条指令位于 `**0x1000 ~ 0x1010**` 地址区间。这是 RISC-V 架构规定的“复位向量地址”，硬件复位后 PC 寄存器会默认指向 `0x1000`，因此初始指令从该地址开始执行。

2. 这些初始指令主要完成了哪些功能？

`0x1000 ~ 0x1010` 区间的初始指令是 RISC-V 硬件启动的“引导过渡代码”，核心功能是完成从硬件复位到加载 bootloader（OpenSBI）的准备工作，具体分为 3 步：

初始化寄存器与地址计算：

`auipc t0, 0x0`：通过“PC 相对寻址”将当前 PC 值（`0x1000`）加载到 `t0`，为后续地址计算提供基准；

`addi a1, t0, 32`：计算得到 `a1=0x1020`，该地址通常用于传递 bootloader 所需的参数（如设备信息）。

获取硬件线程信息：

`csrr a0, mhartid`：读取“机器模式硬件线程 ID 寄存器（`mhartid`）”的值并存入 `a0`，本次调试中 `a0=0`（表示当前仅 1 个硬件线程在运行），该信息用于后续 bootloader 初始化多线程（若有）。

跳转到 `bootloader` 入口：

`ld t0, 24(t0)`：从 `0x1018` 地址（`t0+24`）加载 `bootloader`（OpenSBI）的入口地址（`0x80000000`）到 `t0`；

`jr t0`：通过寄存器跳转指令，将程序控制权移交到 `0x80000000` 处的 OpenSBI，完成初始引导阶段的任务。