操作系统实验报告Lab1

1.实验目标

可执行内核和启动流程

2.实验原理

2.1 分析构建规则和设计原理

2.1.1 构建

```
include tools/function.mk
```

引入引入 function.mk 中定义的一系列宏/函数（如 add_files、create_target、read_packet、totarget 等），用于自动收集源文件、生成对象列表和创建目标规则。

```
KOBJS   = $(call read_packet,kernel libs)

# create kernel target
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $(call symfile,kernel)

$(call create_target,kernel)
```

KOBJS 由 function.mk 的 read_packet/getting 宏生成，包含 kernel 所需的所有 .o 文件。链接步骤使用 $(LD) 并通过 -T tools/kernel.ld 显式指定链接脚本，将 KOBJS 链接为 ELF 可执行内核bin/kernel。随后用 objdump 生成反汇编和符号表文件。

```
UCOREIMG    := $(call totarget,ucore.img)

$(UCOREIMG): $(kernel)
    $(OBJCOPY) $(kernel) --strip-all -O binary $@
$(call create_target,ucore.img)
```

ucore.img 目标是用 objcopy 将 ELF转为纯二进制镜像ucore.img。

我们用qemu生成的镜像是：

```
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000
```

可以看到，qemu把 ucore.img 以 loader 方式装载到物理地址 0x80200000，实现模拟器加载并跳转到 entry.S。

Makefile 中用到的辅助函数和通用规则存放在function.mk中，核心编译模版：

```
define cc_template
$$(call todep,$(1),$(4)): $(1) | $$$$(dir $$$$@)
    @$(2) -I$$(dir $(1)) $(3) -MM $$< -MT "$$(patsubst %.d,%.o,$$@) $$@"> $$@
$$(call toobj,$(1),$(4)): $(1) | $$$$(dir $$$$@)
    @echo + cc $$<
    $(V)$(2) -I$$(dir $(1)) $(3) -c $$< -o $$@
ALLOBJS += $$(call toobj,$(1),$(4))
endef
```

里面有两条规则：1.生成 .d（依赖）规则，调用 gcc -MM 写入 .d；2.生成 .o 规则，编译源文件为目标对象，并把对象追加到 ALLOBJS。

2.1.2 链接

kernel.ld 是我们的链接脚本，指定内核各段放置地址：

```
OUTPUT_ARCH(riscv)
ENTRY(kern_entry)

BASE_ADDRESS = 0x80200000;

SECTIONS
{
    /* Load the kernel at this address: "." means the current address */
    . = BASE_ADDRESS;

    .text : {
        *(.text.kern_entry .text .stub .text.* .gnu.linkonce.t.*)
    }

    ...
    /* Adjust the address for the data segment to the next page */
    . = ALIGN(0x1000);

    .data : { ... }
    ...
}
```

内核的基址常量是0x80200000，`. = BASE_ADDRESS` 将链接脚本的当前位置设置为 BASE_ADDRESS，后续的第一个段（.text）就从该物理地址开始放置，保证最终 ELF/二进制镜像以 0x80200000 为加载起始地址。`. = ALIGN(0x1000);` 在放置完只读段后，把位置对齐到 0x1000（页边界）再放置 .data/.bss，保证数据段从页对齐地址开始。

我们看一下链接后的内核kernel.asm：

```
kern_entry:
    la sp, bootstacktop    "bootstacktop": Unknown word.
    80200000:   00003117            auipc   sp,0x3    "auipc": Unknown word.
    80200004:   00010113            mv  sp,sp

    tail kern_init
    80200008:   a009                    j   8020000a <kern_init>

000000008020000a <kern_init>:
#include <sbi.h>
int kern_init(void) __attribute__((noreturn));    "noreturn": Unknown word.

int kern_init(void) {
    extern char edata[], end[];    "edata": Unknown word.
    memset(edata, 0, end - edata);    "memset": Unknown word.
    8020000a:   00003517            auipc   a0,0x3    "auipc": Unknown word.
    8020000e:   ffe50513            addi    a0,a0,-2 # 80203008 <edata>    "ac
    80200012:   00003617            auipc   a2,0x3    "auipc": Unknown word.
    80200016:   ff660613            addi    a2,a2,-10 # 80203008 <edata>    "a
```

entry 在 0x80200000 设置内核栈指针后做尾跳转到 0x8020000a 的 kern_init,kern_init 建立栈帧，调用 memset 清 bss、调用 cprintf 输出，然后停在无限循环。数据段（.data/.bss）被放在 0x80203000 附近，la/auipc 指令使用 PC 相对偏移访问这些符号。

2.1.3 汇编

设置内核栈指针（sp），并做尾跳转进入 C 语言内核入口函数 kern_init：

```
    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init
```

`la sp, bootstacktop`将标号 bootstacktop （.data段）的地址装入寄存器 sp（栈顶），为后续 C 代码建立初始内核栈（因为RISC-V 约定栈向下增长，sp 应指向栈顶）。

下面是为栈分配的.data段：

```
.section .data
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

3 执行构建流程

3.1 构建阶段

执行make命令：

```
liuliuczxt@LAPTOP-VDR8TECP:/mnt/f/CZXT/labcode/lab1$ make
+ 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
```

写好的Makefile 会编译 kern 下的 .S/.c 文件、libs 下的库、将目标对象通过 kernel.ld链接，生成 ELF 内核镜像（bin/kernel）并用 objcopy 生成二进制镜像ucore.img。

3.2 启动流程（运行时）

执行make qemu命令：

```
liuliuczxt@LAPTOP-VDR8TECP:/mnt/f/CZXT/labcode/lab1$ qemu-system-riscv64 -nographic
 -machine virt -bios default -kernel bin/kernel

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          : 0x0000000080000000-0x0000000080001fff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (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
(THU.CST) os is loading ...
```

在makefile中实际执行的代码是：

```
.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#    $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000
```

使用宏定义$(UCOREIMG) $(SWAPIMG) $(SFSIMG)的函数进行目标文件的构建，然后使用qemu语句进行qemu启动加载内核,我们就把ucore跑起来了。固件/OpenSBI 将内核镜像加载到物理地址 0x80200000 并跳转到该地址，执行 entry.S中的入口符号 kern_entry。

4 练习1：理解内核启动中的程序入口操作

```
la sp, bootstacktop
```

完成的操作：

这条指令是 "Load Address" 的缩写。它将 `bootstacktop` 标签所代表的内存地址加载到栈指针寄存器 `sp` 中。在代码的 `.data` 段，`bootstack` 定义了一块内存空间作为内核的初始栈，而 `bootstacktop` 标签位于这块空间的末尾（最高地址处），因为栈在 RISC-V 架构中是向下生长的。

目的：

为即将开始运行的内核 C 代码设置一个**有效的栈空间**。任何 C 函数的调用都需要使用栈来保存局部变量、函数参数和返回地址。在执行任何 C 函数之前，必须先初始化栈指针 `sp`，让它指向一块可用的内存区域。这条指令就完成了这个至关重要的初始化步骤，确保了接下来调用的 `kern_init` 函数能够正常工作。

```
tail kern_init
```

完成的操作：

`tail` 是一条伪指令，它实现了一个尾调用。在汇编层面，它等同于一条无条件跳转指令（`j kern_init`）。它会直接跳转到 `kern_init` 函数的入口地址开始执行，并且**不会在栈上保存返回地址**。

目的：

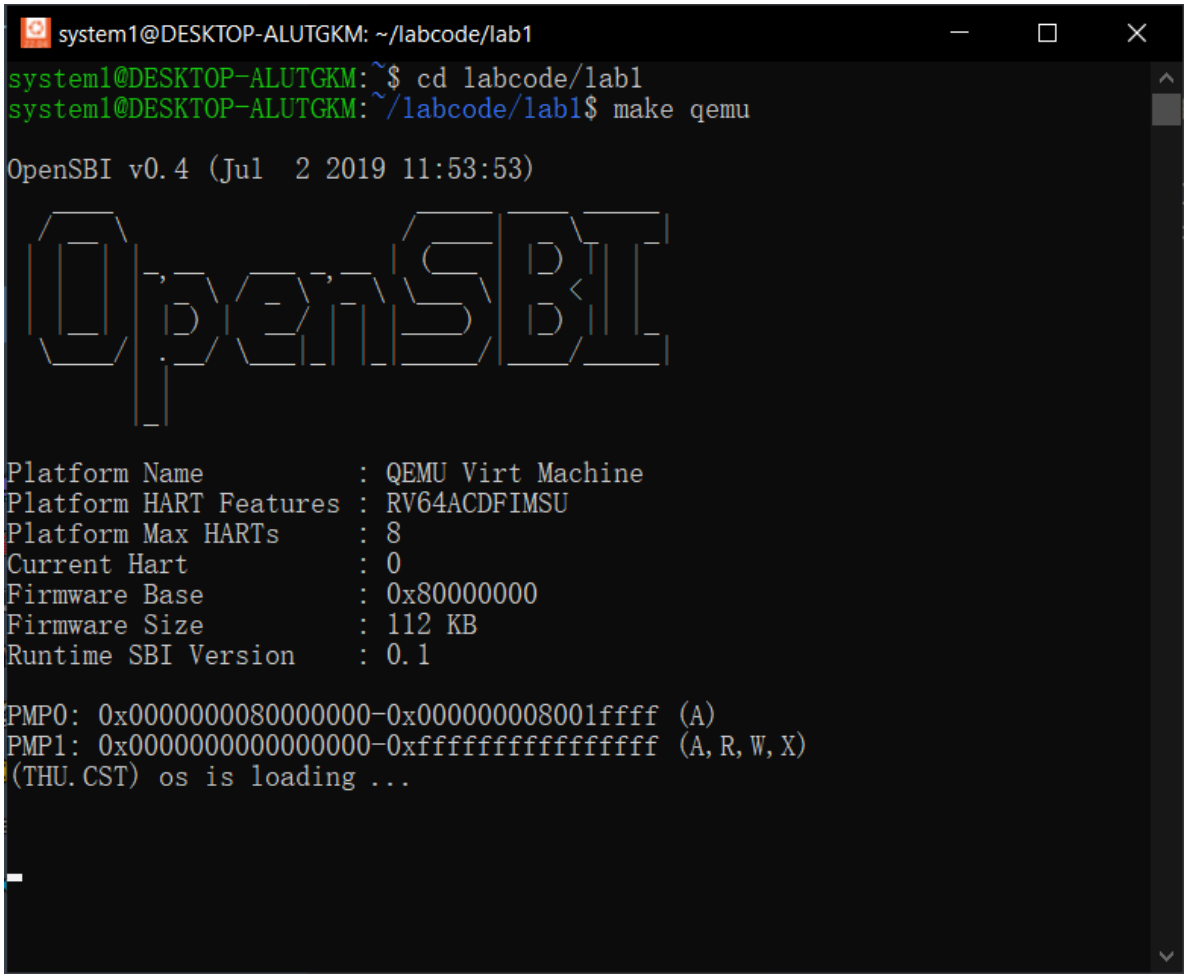
将处理器的控制权从底层的汇编启动代码（`kern_entry`）**永久地**转移给内核的主初始化函数 `kern_init`（这是一个 C 函数）。因为 `kern_entry` 的使命（设置栈指针）已经完成，之后再也不需要返回到这里了，所以使用 `tail`（跳转）是最直接和高效的方式。这标志着内核初始化工作从非常简单的汇编环境过渡到了功能更强大的 C 语言环境。

5 练习2: 使用GDB验证启动流程

5.1 实验内容及目的

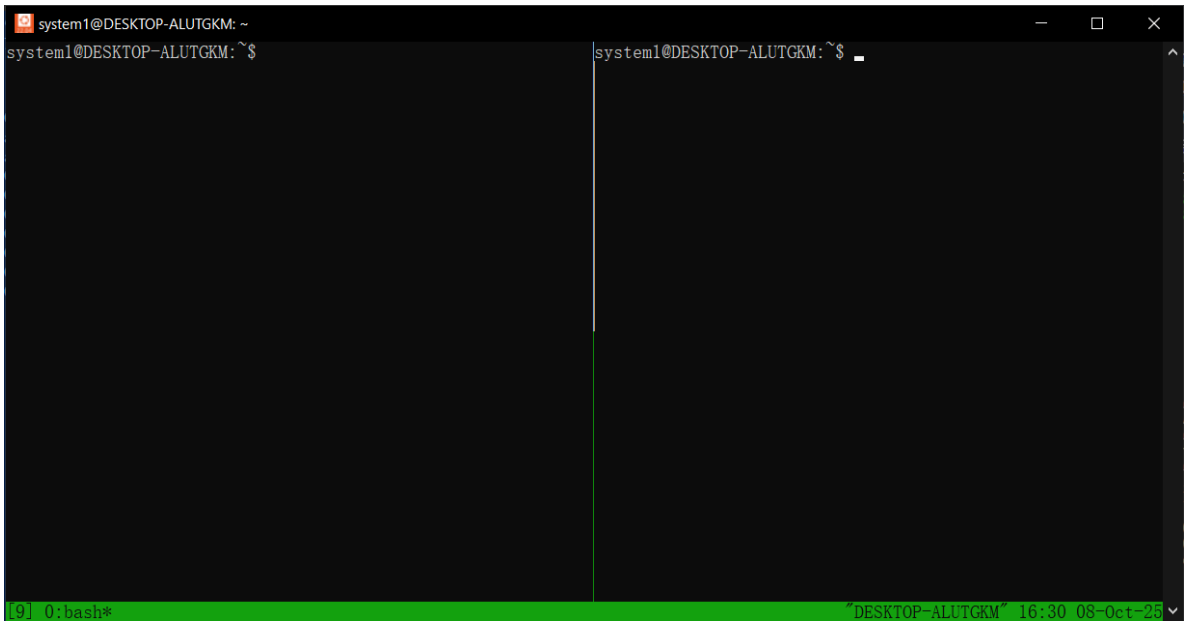
为熟悉使用 QEMU 和 GDB 的调试方法，使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始，直到执行内核第一条指令（跳转到 0x80200000）的整个过程。

最终运行结果：



5.2 调试方法

实验采用指导手册推荐的方法，通过 `sudo apt install tmux` 安装了 tmux 工具，并在输入指令 `tmux` 后依次键入 `Ctrl + B` + `Shift + %`，达到如下的划分窗格界面效果：



此时，在左侧窗格输入 `make debug` 启动 qemu 但处于暂停并等待 GDB 连接的状态；在右侧窗格输入 `make gdb` 启动 GNU Debugger，与左侧窗格启动的 qemu 连接，即可开始调试。

值得一提的是，在原始实验给定的 Makefile 中，可能存在将 `riscv64-unknown-elf-gdb` 工具路径定死的问题，或需要做一些简单的修改以适配外部指令 `make gdb` 直接启动 GDB：

1.
```
GDB     := $(GCCPREFIX)gdb
```

   改为

```
GDB := /opt/riscv/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin/riscv64-unknown-elf-gdb
```

   （替换的是笔者 Ubuntu 中配置的工具路径，因人而异）

2.
```
gdb:
riscv64-unknown-elf-gdb \
```

   改为

```
gdb:
$(GDB) \
```

5.3 调试过程

初始 `$pc = 0x1000`，以此为出发点，使用指令 `x/10i $pc` 调查即将执行的 10 条 RISC-V 汇编指令。结果如下：

```
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0      # t0 = PC + (0x0 << 12) = 0x1000
   0x1004:      addi    a1,t0,32    # a1 = 0x1000 + 32 = 0x1020
   0x1008:      csrr    a0,mhartid  # a0 = mhartid = 0
   0x100c:      ld      t0,24(t0)   # t0 = [t0 + 24] = *(uint64_t *)(0x1018) = 0x80000000
   0x1010:      jr      t0      # 无条件 jump to 0x80000000
   0x1014:      unimp           # 以下是未定义指令
   0x1016:      unimp
   ... ...
```

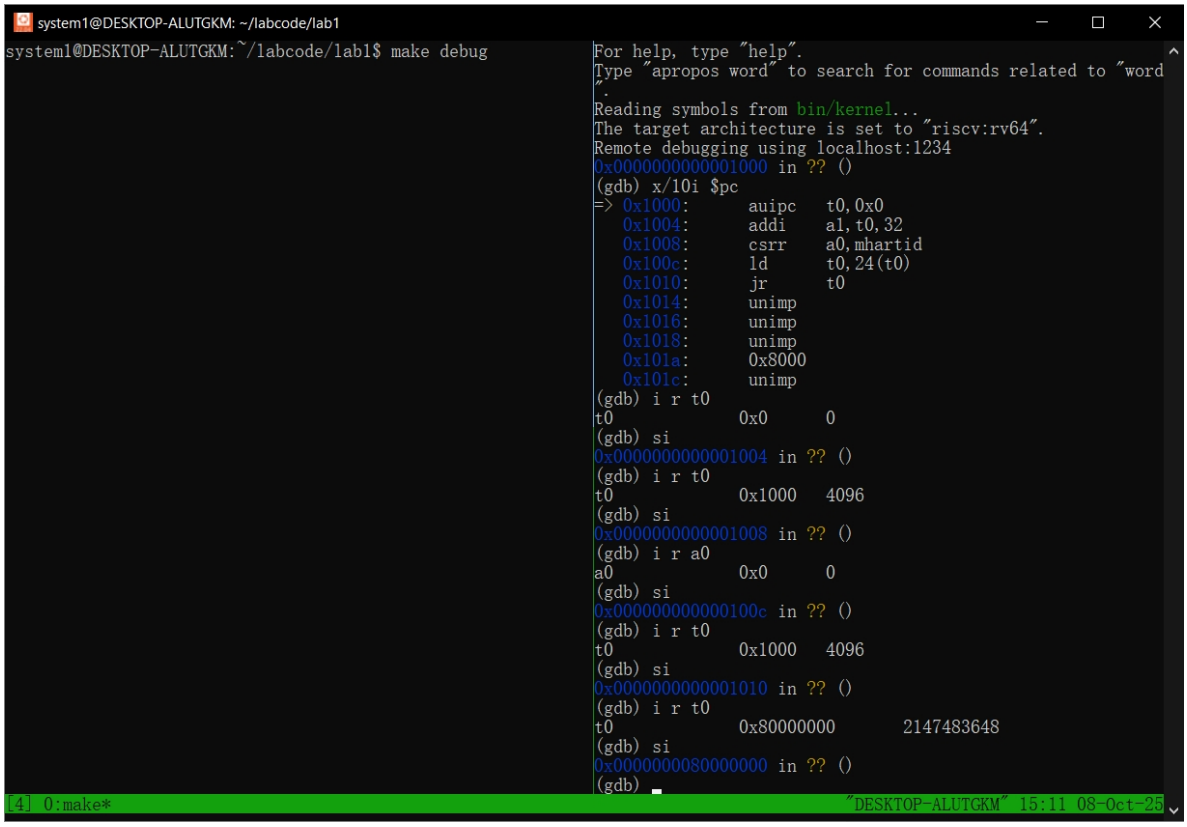
每条指令对应的具体处理都已经展示在上面。

其中 `auipc` = *"Add Upper Immediate to PC"*，在这里字面意义操作是 “把当前 PC 的高 20 位 + `0x0 << 12` 加载到寄存器 `t0`”，那么实际上就是用 `t0` 取当前指令地址的 PC 值。

其中 `csrr` 指令是在从 CSR 寄存器 `mhartid` 读取当前硬件线程的 ID 号，放入 `a0`，此处线程号是 0.

以上汇编代码的主要工作是读取预先存放在内存（或固件）里的跳转目标地址，并跳转过去进行下一步启动，是 CPU 从复位地址（`0x1000`）开始向初始化固件的一个跳板，几乎不需要内部软件的参与。

跳转地址为什么是 `0x80000000`？在下面的逐条指令调试中有直接展示。这里，使用 `si` 指令逐条地单步执行 RISC-V 汇编指令，直到达到 `jr` 跳转指令后的第一步：



过程中除了使用 `si`，还使用了 `i r 寄存器名` 来监视每条指令可能改变的寄存器的值变化，具体分析如下：

```
(gdb) i r t0                // 检查 t0 的值
t0             0x0      0       // 初始为 0
(gdb) si
0x0000000000001004 in ?? ()
(gdb) i r t0
t0             0x1000   4096        // 执行首条指令后取得当前 pc 的值
(gdb) si
0x0000000000001008 in ?? ()
(gdb) i r a0                // 检查 a0 的值
a0             0x0      0       // 原始值 0
(gdb) si
0x000000000000100c in ?? ()
(gdb) i r a0
t0             0x0      0       // 获得当前线程号，即 0
(gdb) si
0x0000000000001010 in ?? ()
(gdb) i r t0
t0             0x80000000       2147483648  // 获得指定地址的值为跳转地址，即 0x80000000
```

可以看到最终通过 `ld` 获得的跳转地址就是 `0x80000000`. 下面就从这个地址为起始做下一轮分析。

同样的道理，使用指令 `x/20i 0x80000000` 调查从跳转地址开始即将执行的 20 条命令，结果如下：

```
(gdb) x/20i 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)
   0x80000028:  auipc   t1,0x0
   0x8000002c:  addi    t1,t1,1016
   0x80000030:  ld      t1,0(t1)
   0x80000034:  auipc   t2,0x0
   0x80000038:  addi    t2,t2,988
   0x8000003c:  ld      t2,0(t2)
   0x80000040:  sub     t3,t1,t0
   0x80000044:  add     t3,t3,t2
   0x80000046:  beq     t0,t2,0x8000014e
   0x8000004a:  auipc   t4,0x0
   ... ...
```

这一部分代码较长，不逐条分析其实际意义。但从整体工程结构上来分析，这一部分应是 qemu 自带的 **bootloader**（引导加载程序）—— **OpenSBI** 固件进行主初始化的汇编代码实现，其核心任务之一是将内核加载到由链接脚本 `kernel.ld` 规定的入口地址（`kern_entry`）。

首先尝试了实验手册建议的 `watch *0x80200000` 指令来观察内核加载瞬间，但发现设置监视点后 `continue` 直接到达了启动程序末尾的死循环，所以放弃了这个跟踪研究方法。由于已经了解了这段汇编代码的作用，或可考虑直接移步至 `kern_entry` 入口之后进行进一步分析。

使用 GDB 设置断点的指令 `b kern_entry`，可以在不知道入口地址的条件下，在目标函数 `kern_entry` 第一条指令处设置断点，得到的输出如下：

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

这就可以看出 `kernel.ld` 规定的入口地址实际上是 `0x80200000`. 使用指令 `c` 即可执行到设置的断点处。 这时再用 `x/10i 0x80200000` 指令调查一下接下来要执行的 10 条 RISC-V 汇编指令，结果如下：

```
(gdb) x/10i 0x80200000
=> 0x80200000 <kern_entry>:     auipc   sp,0x3      # sp = pc + (0x3 << 12) = 0x80200000 + (0x3 << 12) = 0x80203000
   0x80200004 <kern_entry+4>:   mv      sp,sp       # 未改变 sp 的值，猜测是进行了延迟同步
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>  # 无条件跳转到这个函数
   0x8020000a <kern_init>:      auipc   a0,0x3
   0x8020000e <kern_init+4>:    addi    a0,a0,-2
   ... ...
```

如果使用 `list` 来调查此处对应的 `entry.S` 源代码：

```
(gdb) list
4           .section .text,"ax",%progbits
5           .globl kern_entry
6       kern_entry:
7           la sp, bootstacktop
8
9           tail kern_init
10
```

可以发现，这一部分主要做的工作就是将 `bootstacktop` 的地址赋给 `sp` 寄存器作为**栈顶**内存地址，建立**函数栈帧**；然后跳转到 `kern_init` 函数进行内核初始化。（`tail` 是 “尾调用” 伪指令，本质是对函数调用指令 `jal` 的一种特殊优化形式，不再保存返回地址）。

当上面这部分代码全部执行完毕后，左边的 debug 窗格已经有了如下输出：

```
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)
```

则 OpenSBI 已经开始运行。

接下来，再对上面跳转目的函数 `kern_init` 设断点进行研究。同理使用 `b kern_init` 设置断点，得到输出如下：

```
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
```

再用 `c` 运行到断点处。为了分析的简便直观性，这里直接用 `list` 指令来获取 `kern_init` 的源 C 代码：

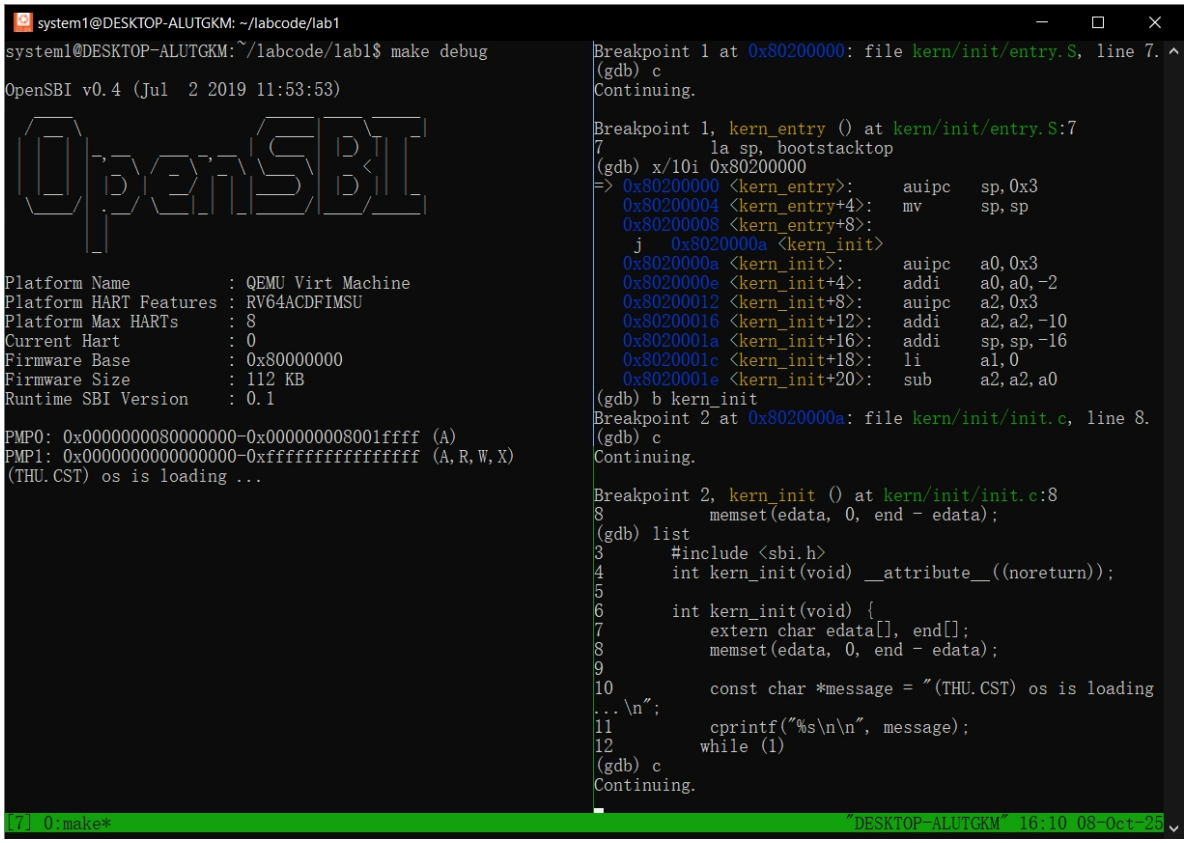
```
(gdb) list
3       #include <sbi.h>
4       int kern_init(void) __attribute__((noreturn));
5
6       int kern_init(void) {
7           extern char edata[], end[];     // 声明两个外部符号
8           memset(edata, 0, end - edata);  // 清零 BSS 段，所有未初始化全局变量
设为 0
9
10          const char *message = "(THU.CST) os is loading...\n";
11          cprintf("%s\n\n", message);     // 调用了自主构造的输出函数
12         while (1)                // 死循环
```

这里两个外部符号之间的地址区间所包含的所有变量，构成了需要设置为 0 的未初始化全局变量集合。

注意这里使用到了 —— 利用 OpenSBI 提供的接口，进行加工、封装等操作创造出的 `cprintf` C 输出函数。

再执行一次 `c`，程序就到达末尾的输出函数，然后进入死循环状态了：

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



调试过程到这里就结束了。

5.4 回答问题

Q1：RISC-V 硬件加电后最初执行的几条指令位于什么地址？

A：RISC-V 硬件加电后最初执行的指令地址位于 0x1000 和 0x1010 之间。

Q2：它们（RISC-V 硬件加电后最初执行的几条指令）主要完成了哪些功能？

A：对每条指令的解析：

```
0x1000:      auipc   t0,0x0 # pc 当前值赋给 t0
0x1004:      addi    a1,t0,32   # t0 加上 32 赋给 a1，为 OpenBSI 参数读取作预备
0x1008:      csrr    a0,mhartid # 获取当前线程 ID 赋给 a0
0x100c:      ld      t0,24(t0)  # 按规则，取 pc 后一定地址处的跳转目的地址
0x1010:      jr      t0     # 无条件跳转至 opensbi.bin 地址
```

总体来看，这几条指令是 CPU 复位后向 bootloader 程序 OpenBSI 的一个**跳板**代码段，为 OpenBSI 读取参数提供预先帮助，让后向 OpenBSI 主程序进行跳转。

6 实验中重要的知识点

6.1 链接脚本（`kernel.ld`）与内存布局

链接脚本 `kernel.ld` 精确控制内核代码段（`.text`）、数据段（`.data`）等被放置在物理内存的哪个具体地址（如 `0x80200000`）。这是确保引导加载程序（Bootloader）能正确加载并跳转到内核入口的前提。

- **对应OS原理：操作系统结构与内存管理**
- **关系**：链接脚本是实现操作系统内存布局这一理论概念的具体工具。OS原理告诉我们，内核必须被加载到内存的特定位置才能工作；而实验中的链接脚本正是告诉链接器"如何"将编译好的各个部分组合起来，并安放到这些"特定位置"的配置文件。

6.2 汇编入口（`entry.S`）与栈指针初始化

在实验中，内核的第一条指令并非 C 代码，而是几行汇编指令。其核心任务是设置栈指针 `sp`（`la sp, bootstacktop`），为即将运行的 C 函数 `kern_init` 准备一个有效的栈空间，然后通过 `tail kern_init` 跳转过去。

- **对应OS原理：系统引导与上下文初始化**
- **关系**：OS原理中的"系统引导"描述了从加电到内核主函数运行的全过程。实验中的 `entry.S` 正是这个过程中的一个关键环节：它负责从机器的原始状态（没有栈）过渡到 C 语言所要求的、结构化的执行环境（有栈）。
- **差异**：原理是宏观的流程描述，而实验中的汇编代码是这个流程中"承上启下"的微观实现。它解决了 C 语言无法解决的"第一个栈从哪里来"的问题。

6.3 通过 GDB 跟踪 QEMU 启动流程

使用 GDB 从硬件加电的初始地址（`0x1000`）开始，单步跟踪了固件（OpenSBI）的执行，观察它如何加载我们的内核镜像到 `0x80200000`，并最终跳转到内核入口 `kern_entry`。

- **对应OS原理：引导加载程序（Bootloader）**
- **关系**：OS原理告诉我们，在内核运行之前，需要一个名为 bootloader 的小程序来完成硬件初始化、加载内核镜像等工作。实验中的 GDB 调试过程，就是对 QEMU 模拟环境中 bootloader（即 OpenSBI）工作流程的一次"亲眼见证"，将抽象的加载过程变得具体可见。
- **差异**：原理是一个功能模块的定义（bootloader 的职责），而实验提供了一种研究和验证该模块具体行为的方法（使用 GDB 跟踪调试）。

6.4 在本实验中未体现的知识点

1. **中断处理**

   在本次实验中，内核启动后就进入了一个死循环 `while(1)`，它没有与外部设备交互，也没有响应任何时钟中断或异常。而在一个真正的操作系统中，中断是驱动整个系统运行的脉搏，是实现多任务调度、I/O 操作和处理错误的基石。

2. **虚拟内存**

   本次实验全程使用物理地址（`0x80200000`）。内核代码直接在物理内存上运行。而现代操作系统最重要的特征之一就是虚拟内存机制，它为每个进程提供独立的地址空间，并负责虚拟地址到物理地址的转换。这部分内容在本次实验中尚未涉及。