OS lab1

成员信息

解子萱 2312585 信息安全、法学双学位班

崔颖欣 2311136 信息安全、法学双学位班

范鼎辉 2312326 信息安全

实验环境

先前做研究的时候指导教师给我们分配了真实物理服务器，因此本组实验将在真实物理服务器上进行，相关信息展示如下：

```
➜ ~ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.5 LTS
Release:        22.04
Codename:       jammy
➜ ~ uname -a
Linux lx 5.15.0-131-generic #141-Ubuntu SMP Fri Jan 10 21:18:28 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
```

实验源码路径为~/OS/lab*

练习1：理解内核启动中的程序入口操作

> 阅读 kern/init/entry.S内容代码，结合操作系统内核启动流程，说明指令 la sp, bootstacktop 完成了什么操作，目的是什么？ tail kern_init 完成了什么操作，目的是什么？

首先，kern/init/entry.S汇编码是整个操作系统内核初始化的入口点，其代码结构主要分为三部分：

- #include 包含指令部分：引入了mmu内存管理单元和memlayout内存布局的头文件，说明接下来的汇编码很可能是在初始化相关内存、堆栈管理任务；
- .section .text 代码段，存放可执行代码；
- .section .data 数据段，定义基本数据；

```
#include <mmu.h>
#include <memlayout.h>

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

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

在分析最重要的.text段代码前，我们先来看看.data段的内容。作为铺垫，它主要完成了两件事情：

1. 通过align实现内存页对齐：我们从mmu.h中找到页大小位移量 `PGSHIFT` 的宏定义（此处PGSIZE页大小设定为4KB，故PGSHIFT的值就是 12），通过 `.align PGSHIFT` 命令确保数据按照页边界对齐。

```
#define PGSIZE          4096                    // bytes mapped by a page
#define PGSHIFT         12                      // log2(PGSIZE)
```

```
#define KSTACKPAGE          2                       // # of pages in kernel stack
#define KSTACKSIZE          (KSTACKPAGE * PGSIZE)   // sizeof kernel stack
```

2. 通过space初始化栈空间：我们从memlayout.h中找到栈空间大小 `KSTACKSIZE` 的宏定义，通过 `.space` 命令为内核栈分配内存空间，两个全局符号bootstack（栈底）和bootstacktop（栈顶）分别指向了该栈空间的两端。

现在我们开始分析.text段， `ax` 表示这个段可读可执行， `%progbits` 表示这个段包含程序实际的指令，它也主要在做两件事情：

1. 栈使用初始化：定义全局符号 `kern_entry`，标志着程序的入口点。操作系统启动时，控制权将转移到kern_entry指向的代码： **`la sp, bootstacktop`**。 `la` 是 `load address` 指令，用于将地址加载到寄存器中； `sp` 是栈指针寄存器，用于存储栈顶位置。**在这里，`bootstacktop` 的地址被加载到 `sp` 中，相当于完成了栈空间准确定位，这意味着位于 `bootstack` 的内存开始正式作为栈被使用！**由于栈从高地址向低地址增长，故这里 `sp` 先存储的是栈顶地址。

2. 内核初始化跳转： **`tail kern_init` 表示将控制权传递给 `kern_init` 函数，并且不会返回，相当于在完成栈初始之后进入到 init 函数，继续执行内核的相关初始化工作。**后边我们知道，这里其实是跳转到了 `init.c` 文件中的函数，它会继续执行内存清零、打印加载信息、死循环等操作。

至此，整个entry.S汇编码的内容分析完毕，总结来说，它主要完成内存页对齐设置、栈空间分配、栈指针寄存器初始化等操作，并在最后实现将控制权转交给 `kern_init` 函数，继续向后执行内核的相关初始化工作。

练习2: 使用GDB验证启动流程

> 为了熟悉使用 QEMU 和 GDB 的调试方法，请使用 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。通过调试，请思考并回答：RISC-V 硬件加电后最初执行的几条指令位于什么地址？它们主要完成了哪些功能？请在报告中简要记录你的调试过程、观察结果和问题的答案。

> **题外话**：ucore指导书中推荐了终端利器tmux，用于在同一个窗口里划分出两个窗格进行使用，但其实用Trae能非常方便地实现这一目的：
>
> 
>
>
> 基于此，我们开始练习2的实验

MROM

首先执行下面两条指令，检查PC寄存器的值是否为0x1000，并观察PC附近的20条指令：

```
(gdb) p/x $pc
$1 = 0x1000
(gdb) x/20i $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
   0x101e:      unimp
```

> 注：为节省篇幅，我们没有完整展示20条指令（因为只有前几条有用）

可以看到，PC确实为0x1000，而PC附近的指令（即CPU上电最初执行的初始化指令）我们解释如下：

代码块

```
(gdb) x/20i $pc
=> 0x1000:      auipc   t0, 0x0             ; 将当前PC的高20位与0左移12位相加，结果存入t0。t0 = PC + 0x0 = 0x1000
   0x1004:      addi    a1, t0, 32          ; a1 = t0 + 32 = 0x1000 + 32 = 0x1020。a1通常用作函数第二个参数
   0x1008:      csrr    a0, mhartid         ; 读取机器硬件线程ID(mhartid)到a0寄存器。a0通常用作函数第一个参数
   0x100c:      ld      t0, 24(t0)          ; 从内存地址(t0 + 24) = (0x1000 + 24 = 0x1018)加载双字到t0
   0x1010:      jr      t0                  ; 跳转到t0寄存器中的地址执行，即0x80000000，OpenSBI固件地址
   0x1014:      unimp                       ; 未实现指令
   0x1016:      unimp                       ; 未实现指令
   0x1018:      unimp                       ; 未实现指令（但注意：0x1018是上条ld指令加载的地址）
   0x101a:      0x8000                      ; 0x8000
   0x101c:      unimp                       ; 未实现指令
```

0x1000处指令将t0赋值为0x1000，0x100c处指令将0x1018地址处的双字加载到t0（即下一条指令跳转的地址），0x1010处指令无条件跳转至t0存储的双字地址处。

这里可能会造成困惑，0x1018地址处明明为unimp，而不是我们想要的0x80000000。仔细观察，0x101a处为0x8000，显然为地址的高四个字节，那么0x1018处应当为低四个字节，即全0。验证一下猜想：

```
(gdb) x/xw 0x1018
0x1018: 0x80000000
```

正确！**那么我们回答一下练习2的问题：RISC-V 硬件加电后最初执行的几条指令位于0x1000~0x1010，这几条指令简单初始化了参数，并且将控制权无条件交给OpenSBI（0x80000000）。**

我们再看看此时的栈：

```
(gdb) x/10x $sp
0x0:    0x00000000      0x00000000      0x00000000      0x00000000
0x10:   0x00000000      0x00000000      0x00000000      0x00000000
0x20:   0x00000000      0x00000000
```

明显未初始化，印证了指导书里的说法：MROM作用十分有限。

OpenSBI

在跳转到0x80000000前，我们执行两条指令，使得每次停止时自动显示当前指令和SP，方便单步观察指令和栈顶变化，然后在0x80000000下断点，最后c执行：

代码块

```
display/i $pc     #方便调试
display/x $sp     #方便调试
b *0x80000000     #下断点
c                 #执行至断点处
```

```
(gdb) c
Continuing.

Breakpoint 1, 0x0000000080000000 in ?? ()
1: x/i $pc
=> 0x80000000:  csrr    a6,mhartid
2: /x $sp = 0x0
```

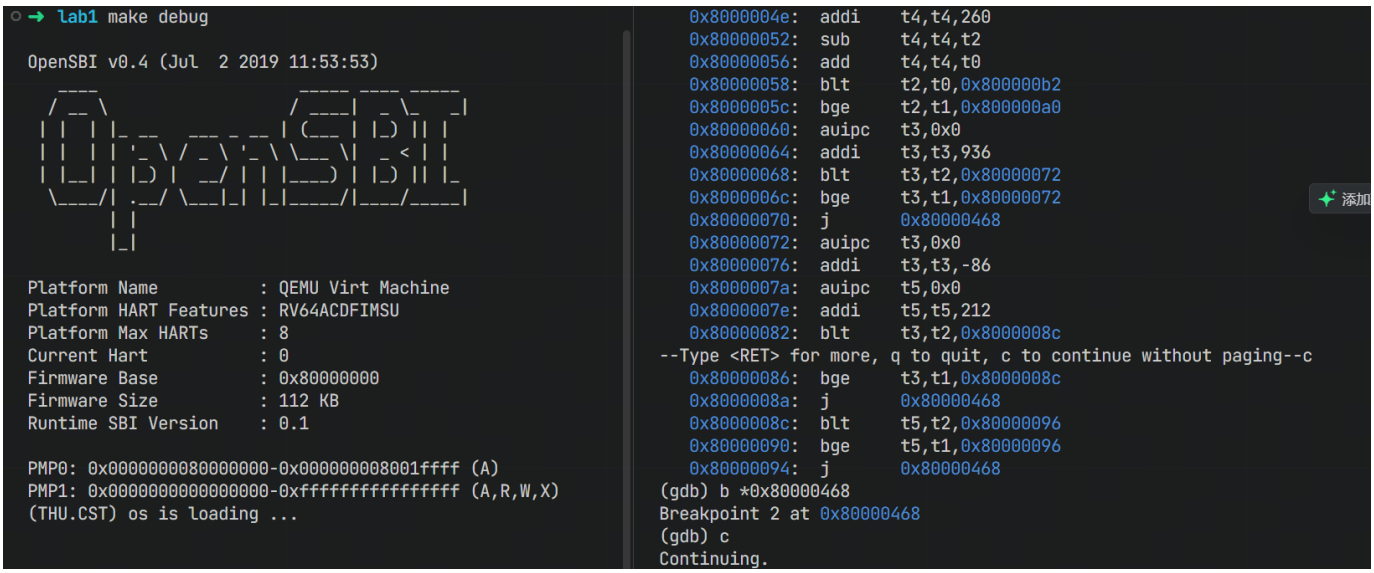
可以看到，已在0x80000000处中断。

观察PC附近的40条指令（这次多看一点）：

```
(gdb) x/40i $pc
=> 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
```

看到了一堆乱七八糟的指令，大模型辅助后了解到，这些指令其实就是在解析函数入口地址表，做一些内存范围检查和初始化。其实也不难理解，因为**OpenSBI本质是函数库**，所以肯定要有对应的初始化操作。

没找到心心念念的跳转到0x80200000的指令，继续探索，发现有好几条汇编指令都跳转到0x80000468，尝试下断点跳转到那里，结果发现左边终端直接输出语句了：



查资料发现，0x80000468处其实是出错后的统一处理，思路错误。只能重新下断点，si单步执行，直到~~跳转到0x80200000~~，结果si了一个世纪也没跳转到0x80200000（似乎指令一直在循环执行）。。。

但我还是想找到跳转到0x80200000的那条指令，在大模型的辅助下，我想到可以反汇编OpenSBI的可执行文件（.bin），找到对应指令。那我们要做的就是**找到.bin文件->objdump反汇编->找到对应指令**。前两步如下图：

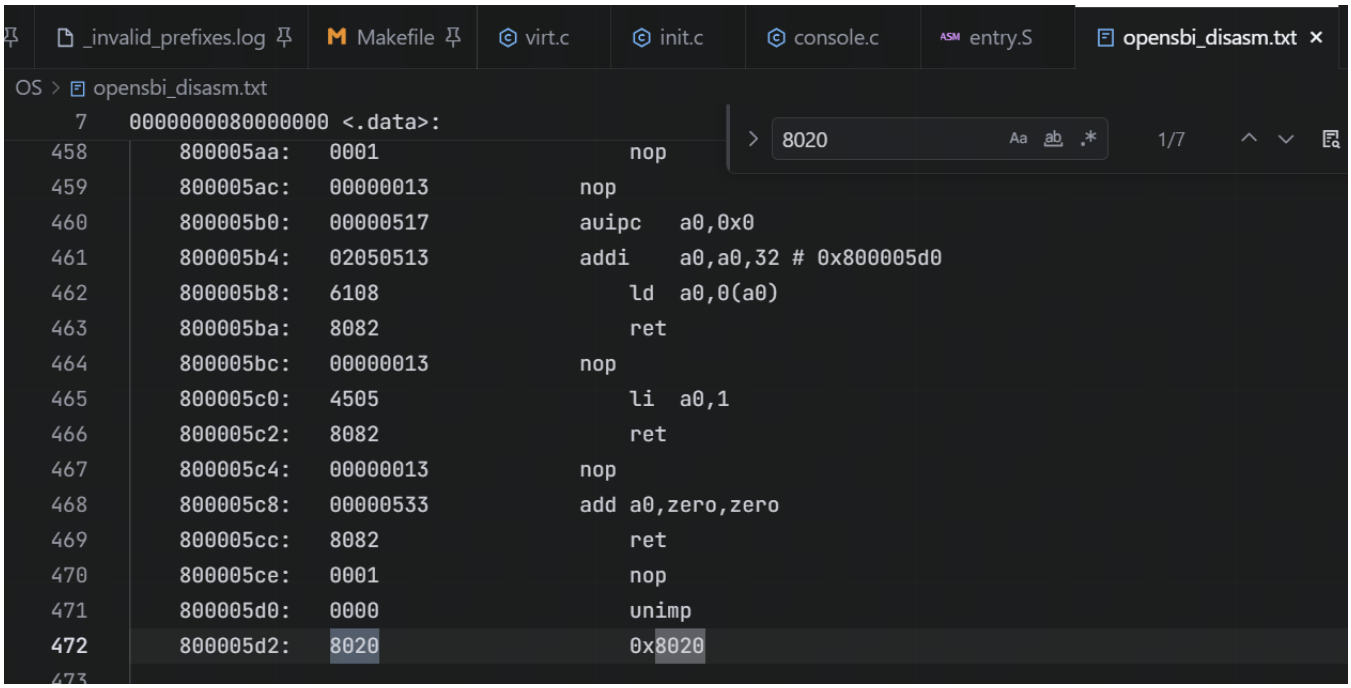
```
➜ OS find . -name "*.bin" | grep riscv

./RISC-V/qemu-4.1.1/pc-bios/opensbi-riscv32-virt-fw_jump.bin
./RISC-V/qemu-4.1.1/pc-bios/opensbi-riscv64-virt-fw_jump.bin
./RISC-V/qemu-4.1.1/pc-bios/opensbi-riscv64-sifive_u-fw_jump.bin
➜ OS riscv64-unknown-elf-objdump -D -b binary -m riscv:rv64 --adjust-vma=0x80000000
./RISC-V/qemu-4.1.1/pc-bios/opensbi-riscv64-virt-fw_jump.bin > opensbi_disasm.txt
```

可以看到，我们找到了.bin文件，并且反汇编保存到.txt文件中。

> 注：objdump可以直接反汇编.elf文件，但对于.bin原始二进制文件，需要指定架构和起始地址。

查找.txt文件：



终于找到0x80200000的相关信息了！可以看到，上图中0x800005b8处指令将0x80200000赋值给a0，后续应该是调用了某个函数，a0的值作为第一个参数传入，达成跳转到0x80200000的目的。这里我们不再追踪后面的数据流。

那么我们继续回答练习2的问题：OpenSBI指令从0x80000000，指令解析了自带库函数入口地址表，做了一些内存范围检查和初始化，并且把操作系统内核镜像加载到物理内存地址 `0x80200000` ，最终跳转到内核入口。

kernel

按照实验指导书的提示，接下来是内核启动执行，执行 `kern/init/entry.S` 。我在 `0x80200000` 位置设断点，查看20条汇编指令如下：

```
(gdb) x/20i $pc
=> 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
   0x80200020 <kern_init+22>:   sd      ra,8(sp)
   0x80200022 <kern_init+24>:
    jal ra,0x802004b6 <memset>
```

前两句栈指针SP设置到 bootstacktop，因为内核启动时还没有 C 语言栈，所以要手动设置SP；接下来跳转到 C 函数 kern_init（虽然0x8020000a就在下一个地址）

这刚好对应 `entry.S` 文件前两句编译后的机器码，证实在 `0x80200000` 这个地址上存放的是 `kern/init/entry.S` 文件编译后的机器码，这是因为链接脚本将 `entry.S` 中的代码段放在内核镜像的最开始位置。`entry.S` 设置内核栈指针，为C语言函数调用分配栈空间，准备C语言运行环境，然后按照RISC-V的调用约定跳转到 `kern_init()` 函数。

kern/init/entry.S

```
1   kern_entry:
2       la sp, bootstacktop
3       tail kern_init
```

```
(gdb) i r
ra             0x80000a02       0x80000a02
sp             0x8001bd80       0x8001bd80
gp             0x0      0x0
tp             0x8001be00       0x8001be00
t0             0x80200000       2149580800
```

```
(gdb) i r
ra             0x80000a02       0x80000a02
sp             0x80203000       0x80203000 <SBI_CONSOLE_PUTCHAR>
gp             0x0      0x0
tp             0x8001be00       0x8001be00
t0             0x80200000       2149580800
```

再看一眼寄存器的变化情况，除了pc的变化，sp在执行完 `la sp, bootstacktop` 就变成了0x80203000，代码里没有这个数为什么会这样呢?

这句的意思是将符号 `bootstacktop` 的地址加载到寄存器 `sp` 中。因此， `sp` 的值取决于符号 bootstacktop 的地址。借助大模型发现是 `.data` 段中定义了内核启动栈：

代码块

```
1       .align PGSHIFT
2       .global bootstack    #标记栈空间起始地址
3   bootstack:
4       .space KSTACKSIZE    #为内核栈分配一段连续内存空间
5       .global bootstacktop    #标记栈空间结束（顶部）地址
6   bootstacktop:
```

内核的内存布局由链接脚本（`kernel.ld`）决定，我们的链接脚本如下：

代码块

```
1   BASE_ADDRESS = 0x80200000;
2
3   SECTIONS {
4       . = BASE_ADDRESS;
5       .text : { *(.text.kern_entry .text .stub .text.* .gnu.linkonce.t.*) }
6       .rodata : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
7
8       . = ALIGN(0x1000);
9
10      .data : { *(.data) *(.data.*) }
11      .sdata : { *(.sdata) *(.sdata.*) }
```

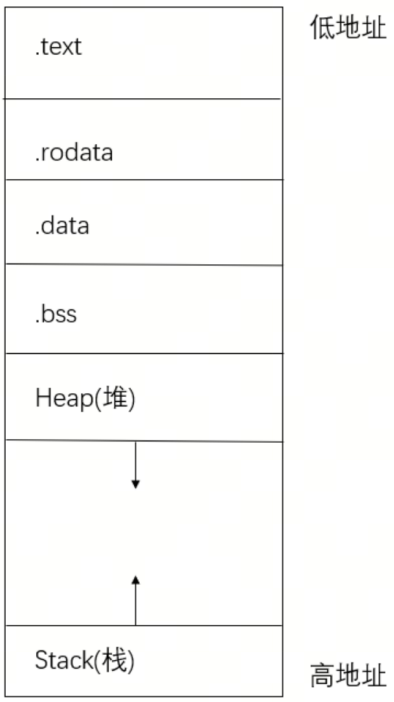
```
12        .bss : { *(.bss) *(.bss.*) *(.sbss*) }
13    }
```

`.text` 段从 **0x80200000** 开始，`.rodata` 在其后，`.data` 段起始地址经过 `ALIGN(0x1000)`（页对齐，4KB），`.bss` 在 `.data` 之后。

RISC-V 架构中栈空间是**从高地址向低地址增长**的，因此，在初始化时将 `sp` 设为 `bootstacktop`，表示当前栈为空，后续函数调用时再逐步向低地址分配栈帧。

综上，整个内存的结构可以被划分成以下表格：

段名	说明	起始地址	结束地址（近似）
.text	内核代码	0x80200000	~0x80202000
.rodata	只读数据	~0x80202000	~0x80202C00
.data	栈定义起始	0x80203000	...
bootstack	栈底	0x80203000 - KSTACKSIZE	
bootstacktop	栈顶	0x80203000	



再用命令证实一下bootstack和bootstacktop的值：

```
(gdb) p &bootstack
$1 = (<data variable, no debug info> *) 0x80201000
(gdb) p &bootstacktop
$2 = (<data variable, no debug info> *) 0x80203000 <SBI_CONSOLE_PUTCHAR>
```

综上，这一阶段主要由 `entry.S` 完成。它的任务是为即将运行的 C 语言代码建立最小执行环境，包括：

1. **加载内核栈指针**（`la sp, bootstacktop`，将符号 `bootstacktop` 对应的地址加载到 `sp`，设置栈顶指针）
2. **跳转到内核主函数**（`tail kern_init`，等价于 `jalr x0, kern_init`，跳转到 `kern_init` 并丢弃返回地址，相当于无返回的函数调用，意味着 CPU 的执行权由汇编转交给 C 层。

好了，接下来会到地址从 0x8020000a 开始，对应 `kern/init/init.c` 生成的汇编。

kern/init/init.c

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);
   while (1)
        ;
}
```

函数的作用主要有两部分：（1）使用 `memset` 将 `.bss` 段清零（将内核未初始化的全局变量区置为 0）；（2）使用 `cprintf` 打印内核启动提示信息。最后进入死循环，防止返回。

首先第一句中 `memset` 是标准库函数，其参数依次为目标地址、填充值和长度。
在进入函数前，编译器会通过寄存器传递参数：

- `a0 = edata` （.bss 段首地址）`a1 = 0` （填充满0）`a2 = end - edata` （清零长度）

接着通过 `jal ra, memset` 调用函数。

```
(gdb) x/20i $pc
=> 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
   0x80200020 <kern_init+22>:   sd      ra,8(sp)
   0x80200022 <kern_init+24>:   jal     ra,0x802004b6 <memset>
```

此时 `ra` 被更新为返回地址；`sp` 在函数内部会被向下移动（分配栈空间）；临时寄存器 `t0~t2` 在循环写入时被频繁使用。

```
(gdb) i r a0 a1 a2 ra sp
a0             0x80203008       2149593096
a1             0x82200000       2183135232
a2             0x80203012       2149593106
ra             0x80000a02       0x80000a02
sp             0x80203000       0x80203000 <SBI_CONSOLE_PUTCHAR>
```

综上，第一句清空 `.bss` 段，即所有未初始化的全局变量区域。其本质是在系统启动阶段建立“确定的内存状态”。

> **原理对应：**

> - 操作系统原理中，“内核内存布局”要求 `.bss`、`.data`、`.text` 各段独立对齐，保证全局变量、代码和常量的独立性；
> - 实验中，`memset()` 通过逐字节写零，保证所有未初始化全局变量均为零，避免因内存垃圾导致的异常。

在汇编层面，`memset` 会展开为循环指令，每次写入 8 字节数据至目标地址，并更新计数寄存器。GDB 观察时可发现寄存器 a0（目标地址）、a2（计数长度）递减变化。

接下来是函数定义字符串常量，并调用 `cprintf` 进行输出，而这个函数并非直接属于 C 标准库，而是我们自己在内核环境中实现的输出接口。 在没有操作系统支持的环境中，我们无法使用标准 I/O 函数，因此需要从最底层重新构建一套输出系统。

整体的调用链条如下

代码块

```
kern_init()
  → cprintf()
  → vcprintf()
  → vprintfmt()
  → cputch()
  → cons_putc()
  → sbi_console_putchar()
  → sbi_call()
  → ecall 指令（陷入 M 模式，由 OpenSBI 执行）
```

整个调用链的关键在于 `sbi_call()` 函数，它通过**内联汇编**手动构造参数并触发特权级陷入，使得 S 模式的内核可以调用 M 模式的 SBI 服务。

`sbi_call()` 通过内联汇编执行 `ecall` 指令，实现从 S 模式到 M 模式的受控跳转如下：

代码块

```
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n" // 功能编号 → a7
        "mv x10, %[arg0]\n"     // 参数0 → a0
        "mv x11, %[arg1]\n"     // 参数1 → a1
        "mv x12, %[arg2]\n"     // 参数2 → a2
        "ecall\n"               // 环境调用
        "mv %[ret_val], x10"    // 返回值从a0取回
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0),
```

```
            [arg1] "r" (arg1), [arg2] "r" (arg2)
          : "memory"
      );
      return ret_val;
  }
```

在 `sbi_console_putchar()` 中，SBI 调用编号为 SBI_CONSOLE_PUTCHAR = 1，用于输出一个字符：

代码块

```
void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}
```

这意味着每一次我们在屏幕上看到字符输出时，实际上都经历了一次从 S 模式 → M 模式 → S 模式 的陷入与返回过程。

> **陷入调用机制：**
>
> - **参数传递：** a0–a2 传递调用参数，a7 传递功能号（例如 `SBI_CONSOLE_PUTCHAR = 1`）。
> - **控制流：** 执行 `ecall` → 切换到 M 模式 → 执行 SBI Handler → 返回 S 模式。
> - **返回值：** 从 a0 寄存器带回。

```
(gdb) b sbi_console_putchar
Breakpoint 1 at 0x80200480: file libs/sbi.c, line 18.
(gdb) c
Continuing.

Breakpoint 1, sbi_console_putchar (ch=40 '(') at libs/sbi.c:33
33          sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
(gdb) info registers a0 a1 a2 a7 pc
a0             0x28     40
a1             0x80202f94       2149592980
a2             0x73     115
a7             0x0      0
pc             0x80200480       0x80200480 <sbi_console_putchar>
(gdb) si
0x0000000080200482 in sbi_call (arg2=0, arg1=0, arg0=40, sbi_type=1) at libs/sbi.c:18
18          __asm__ volatile (
(gdb) info registers a0 a1 a2 a7 pc
a0             0x28     40
a1             0x80202f94       2149592980
a2             0x73     115
a7             0x0      0
pc             0x80200482       0x80200482 <sbi_console_putchar+2>
```

在更高层的 `stdio.c` 文件中，`cputch()` 将每个字符传入 `cons_putc()`：

代码块

```
static void cputch(int c, int *cnt) {
    cons_putc(c);
    (*cnt)++;
}
```

而 `cons_putc()` 只是对 `sbi_console_putchar()` 的简单包装：

代码块

```
1  void cons_putc(int c) {
2      sbi_console_putchar((unsigned char)c);
3  }
```

这层层封装的结构使得上层的 `cprintf()` 可以像标准库函数一样调用，却能在最原始的硬件环境下输出字符。
当我们运行内核后，屏幕输出如下字符串，实际上是由几十次 `ecall` 完成的每个字符输出。

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

回到 `init.c`，末尾的 `while(1);` 被编译为：`j .`，即不停跳回自身地址形成死循环。此时所有寄存器值保持不变，只有 `pc` 在同一地址重复执行。 GDB 单步 (`si`) 时会发现程序永远停留在该位置，不再有任何状态变化。

通过以上观察可知，在整个 `kern_init()` 执行过程中，寄存器的变化主要集中在函数调用阶段。参数寄存器 (`a0~a2`) 和返回地址寄存器 (`ra`) 在函数调用前后不断更新，而栈指针 (`sp`) 的值随函数调用深度动态变化。

`memset` 和 `cprintf` 两个调用展示了典型的函数调用机制与栈空间分配方式，也验证了 RISC-V 调用约定（a0–a7 传参，ra 保存返回地址，sp 管理栈）。最终，程序停在死循环处，等待后续调度逻辑建立，说明内核的最小初始化已完成。

重要知识点总结

一、实验中的核心知识点及其与OS原理的对应关系

1. 启动引导过程与特权级切换

实验中我们观察到了完整的三段式启动：MROM(0x1000) → OpenSBI(0x80000000) → Kernel(0x80200000)。这个过程体现了RISC-V架构的特权级模型：M模式（机器模式）→ S模式（监管者模式）。

OS原理中讲的是"bootloader → kernel"的两阶段启动，但在RISC-V架构下，OpenSBI充当了一个中间层的角色。它不仅完成了bootloader的工作（加载内核），还提供了SBI接口作为S模式和M模式之间的通信桥梁。这种设计让内核不需要直接处理硬件细节，提高了可移植性。

从原理上看，特权级的存在是为了保护系统资源不被任意访问。M模式拥有完全的硬件访问权限，S模式只能通过SBI调用来请求M模式的服务。实验中每次调用 `sbi_console_putchar` 都会触发一次 `ecall` 指令，这就是一次完整的特权级切换过程。

2. 函数调用约定与栈管理

实验中通过 `la sp, bootstacktop` 初始化栈指针，这是建立C语言运行环境的第一步。我们看到RISC-V使用a0-a7传参、ra保存返回地址、sp管理栈空间的规则。

OS原理强调进程需要独立的栈空间来保存局部变量和函数调用上下文，但原理课通常不会深入到汇编层面去讲调用约定。实验让我们看到了这些抽象概念的底层实现：每次 `jal` 指令会自动将返回地址存入ra，每次函数调用都会通过 `addi sp, sp, -xxx` 来分配栈帧空间。

这里有个细节值得注意：栈是从高地址向低地址增长的，所以初始化时sp指向bootstacktop（栈顶），后续通过减小sp来分配空间。这种设计在原理课上不会细讲，实际观察汇编代码能更深刻理解其合理性。

3. 内存布局与链接过程

实验通过链接脚本 `kernel.ld` 明确了内核各段的布局：.text从0x80200000开始，.rodata、.data依次排列，.bss紧随其后。这个布局不是随意的，而是精心设计的结果。

OS原理中讲进程地址空间时会提到代码段、数据段、BSS段等概念，但通常是从虚拟地址空间的角度讲的。实验中我们看到的是物理地址空间的实际布局，还没有引入虚拟内存机制。`memset(edata, 0, end - edata)` 这行代码的作用就是把BSS段清零，确保未初始化的全局变量都是0值，而不是内存中的随机数据。

链接脚本的 `.align PGSHIFT` 指令强制4KB对齐，这是为后续引入页表做准备的。虽然当前还在物理地址上运行，但内核镜像的布局已经按照页的边界来组织了。

4. SBI接口与硬件抽象

实验中最有意思的部分是输出字符的实现。从 `cprintf` 到最终的 `ecall` 指令，经历了多层封装：cprintf → vcprintf → vprintfmt → cputch → cons_putc → sbi_console_putchar → sbi_call → ecall。

这体现了OS原理中"分层设计"的思想。每一层都有明确的职责：最上层提供格式化输出接口，中间层负责字符处理和缓冲，底层通过SBI调用陷入M模式访问硬件。这种设计让内核代码不需要关心具体是什么串口硬件，只需要调用统一的SBI接口。

原理课讲设备驱动时会提到硬件抽象层的概念，SBI就是RISC-V架构下的一种硬件抽象实现。通过在M模式提供统一接口，不同的硬件平台可以有不同的OpenSBI实现，但对内核来说接口是一致的。

5. 从汇编到C的过渡

`entry.S` 的核心作用就是建立最小的C运行环境。汇编代码只做了两件事：设置栈指针、跳转到C函数。但这两件事是必不可少的，因为C语言的函数调用、局部变量都依赖栈机制。

OS原理通常假设C环境已经就绪，但实际上在内核最开始启动时，连栈都没有。实验让我们看到了这个"从无到有"的过程。`tail kern_init` 使用的是尾调用优化，它等价于 `jalr x0, kern_init`，即跳转后不保存返回地址，因为这是单向的控制转移，内核初始化函数永远不会返回。

二、实验中体现但原理课较少涉及的知识点

1. 链接脚本的作用

课上老师没有特别详细讲链接过程，但实验中 `kernel.ld` 决定了整个内核镜像的内存布局。链接器根据这个脚本把各个目标文件组合成最终的可执行文件，并确定每个段的起始地址。这是操作系统能够正确加载运行的前提。

2. 启动时的固件角色

RISC-V架构下OpenSBI的存在是个很有特色的设计。它既是bootloader又是运行时服务提供者。相比x86的BIOS/UEFI，OpenSBI更加现代和标准化。实验让我们理解了固件不只是"把内核加载到内存"那么简单，它还要提供持续的运行时支持。

3. 内联汇编的使用

`sbi_call` 函数中的内联汇编展示了如何在C代码中嵌入汇编指令来完成特殊操作。这在内核开发中很常见，因为有些操作（如特权指令、CSR寄存器访问）只能通过汇编完成。原理课通常不会涉及这种实现细节。

三、原理课中重要但实验尚未涉及的知识点

1. 中断与异常处理机制

原理课花了大量篇幅讲中断向量表、中断处理流程、上下文保存与恢复等内容。但当前实验中还没有配置中断处理，系统无法响应任何中断或异常。如果现在发生页错误或时钟中断，系统会直接崩溃。

2. 虚拟内存与页表映射

这是原理课的核心内容之一，但实验一完全运行在物理地址空间。当前内核访问0x80200000就是真实的物理地址0x80200000，没有经过地址转换。

虚拟内存带来的好处是巨大的：进程地址空间隔离、内存保护、按需分页等。实验中虽然用了页对齐，但还没有真正建立页表。开启MMU、配置页表项、处理TLB等内容应该在后续实验中出现。

3. 进程管理与调度

原理课讲了进程的概念、PCB的结构、进程状态转换、各种调度算法等。但实验一的内核只有一个执行流，就是从 `kern_init` 开始一直运行到死循环。没有进程的概念，也就谈不上调度。

进程切换涉及保存和恢复寄存器现场、切换页表、更新调度数据结构等操作。这些都是OS的核心功能，但在启动阶段还不需要。

4. 同步与互斥机制

信号量、互斥锁、条件变量这些同步原语在原理课中占了很大比重，用于解决并发访问共享资源的问题。但实验一是单核单线程执行，不存在并发竞争，所以完全不需要同步机制。

5. 设备驱动模型

虽然实验中通过SBI接口访问了串口输出，但这不是一个完整的驱动程序。真正的驱动需要处理设备注册、中断响应、DMA配置、缓冲区管理等复杂工作。原理课讲的设备管理内容在实验一中基本没有体现。

四、我们的理解与思考

通过这次实验，我们最大的收获是理解了操作系统不是凭空出现的，而是从最底层一步步构建起来的。原理课上学的那些抽象概念，在实现层面都有具体的汇编指令、数据结构、硬件机制与之对应。

比如"进程需要独立的栈空间"这句话，在看到 `la sp, bootstacktop` 之前只是个抽象概念，但当我单步调试看到sp寄存器的值从0变成0x80203000，再看到每次函数调用sp都在变化，这个概念就变得具体可感了。

另一个深刻体会是分层设计的重要性。从 `cprintf` 到 `ecall` 的调用链看起来很长，但每一层都有清晰的职责。这种设计让代码更易于维护和扩展，也是大型软件工程的基本方法论。

实验一只是个开始，很多核心功能还没有实现。但通过观察这个最小可运行内核的启动过程，我对操作系统的整体架构有了更清晰的认识。期待后续实验逐步补全中断、内存管理、进程调度等模块，最终形成一个功能完整的OS内核。

> **模仿宫老师的ucore指导书一句话总结：**
> 从 0x1000 到 0x80200000，是硬件的世界；
> 从 `entry.S` 到 `kern_init()` ，是操作系统诞生的起点。