

操作系统实验一

学号：2313501 姓名：杨楠欣；学号：2213230 姓名：向宇涵；学号：2311366 姓名：邵莫涵

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？

1. 解读指令 `la sp, bootstacktop`

完成的操作

`la` 作为 RISC-V 中用于加载地址的伪指令，会将符号 `bootstacktop` 的物理地址加载到 `sp` 寄存器——栈指针寄存器，完成内核栈指针的初始化。

Note

`bootstack` 是一块大小为 `KSTACKSIZE` 的内存区域，作为内核栈的存储空间。

`bootstacktop` 是该内存区域的顶部地址，栈从高地址向低地址增长。

```
0000000080203000 bootstacktop
```

目的

为内核代码提供第一块可用的栈空间。

2. 解读指令 `tail kern_init`

完成的操作

`tail` 作为 RISC-V 中用于跳转的伪指令，完成的操作是跳转到 `kern_init` 函数并执行。

Note

它的特殊之处在于尾调用优化。

在 `kern_entry` 中，`kern_init` 是最后一条指令，调用后，没有任何后续指令需要执行。这种情况下，保存返回地址完全是多余的。

`tail` 指令的特殊之处在于不保存返回地址和复用当前栈帧，既减少栈空间占用，又提高了执行效率。

目的

将执行权从汇编代码交给 C 代码。

- `kern_init` 定义在 `kern/init/init.c` 中，是内核的C语言入口点，负责完成更复杂的初始化工作。
 - 初始化内核环境：例如，清理 `.bss` 段，为未初始化的全局变量赋零值。
 - 向用户提供可视化反馈：这是操作系统与开发者/用户的第一次交互，通过输出信息告诉我们：“os is loading ...”。

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？（对该问题的具体回答放在了最后）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。

编译器配置

此次实验在wsl中进行，下载的编译器版本为`riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14`，在`.bashrc`文档最后配置上环境变量，输入命令`riscv64-unknown-elf-gcc -v`，可以看到gcc配置成功，gcc版本为10.2.0。

```
export PATH=/mnt/d/RISCVV/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin:$PATH
Supported LTO compression algorithms: zlib
gcc version 10.2.0 (SiFive GCC-Metal 10.2.0-2020.12.8)
```

模拟器配置

因为模拟器配置过高会无法正常进行实验，因此这里单独下载了4.1.1版本的qemu进行实验。下载好后，因为链接本地的文件会出现权限不够的情况，这里需要先把qemu文件拷贝到wsl的本地文件中，再把本地文件的路径配置到环境变量中，可以看到此时的版本是符合要求的。

```
export PATH=$HOME/qemu-4.1.1-install/bin:$PATH
```

```
xyh@TaroHan:~$ qemu-system-riscv64 --version
QEMU emulator version 4.1.1
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
```

1. 在源代码的根目录下执行 `make qemu`

[illegible]

观察到输出一行提示信息 (THU.CST) os is loading, 然后进入死循环。

既然观察到了这个输出信息，也就是说完成了

加电复位 → CPU从0x1000进入MROM → 跳转到0x80000000(OpenSBI) → OpenSBI初始化并加载内核到0x80200000 → 跳转到entry.S → 调用kern_init() → 输出信息 的全过程。

2. 新建终端一（左侧）

执行 `make debug`，启动 QEMU 调试模式。

注：执行后，终端一无任何输出，这是正常现象。因为 -S 选项让 QEMU 一启动就暂停，等待 GDB 连接。

3. 新建终端二（右侧）

- ### 1. 验证 CPU 复位地址 (0x1000)

执行 `make gdb` 启动并连接 GDB。

```
osnanci@LAPTOP-UHMOLN9T:~/操作系统作业/lab1$ make debu
g
osnanci@LAPTOP-UHMOLN9T:~/操作系统作业/lab1$ make gdb
riscv64-unknown-elf-gdb \
-ex 'file bin/kernel' \
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
GNU gdb (GDB) 16.3.90.20250610-git
Copyright (C) 2024 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 redi
stribute 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-pc-linux-gnu
--target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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 ?? ()
(gdb)
```

从当前 GDB 输出 `0x00000000000001000 in ?? ()` 可以确认，程序已成功停在 RISC-V 处理器上电后的初始地址 `0x1000`。

2. 验证 OpenSBI 加载地址（`0x80000000`）和内核加载地址（`0x80200000`）

在 GDB 中设置 OpenSBI 入口断点，跟踪固件执行：

```
osnanci@LAPTOP-UHMOLN9T:~/操作系统作业/lab1$ make debu
g
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)
[

osnanci@LAPTOP-UHMOLN9T:~/操作系统作业/lab1$ make debu
g
-ex 'target remote localhost:1234'
GNU gdb (GDB) 16.3.90.20250610-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gn
.org/licenses/gpl.html>
This is free software: you are free to change and red
stribute 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-pc-linux-gn
--target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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 ?? ()
(gdb) b* kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, li
ne 7.
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb)
```

如上图，左侧终端中，OpenSBI的启动日志明确显示：

- Firmware Base : 0x80000000

这直接证明OpenSBI固件被正确加载到了物理地址0x80000000，与指导书上讲到的QEMU复位代码约定的OpenSBI加载地址完全一致。

此外，日志中还包含其他关键信息，我们可以了解一下：

- Platform Name : QEMU Virt Machine : 确认运行在QEMU模拟的RISC-V虚拟平台。
- Firmware Size : 112 KB : OpenSBI固件的体积。
- Runtime SBI Version : 0.1 : OpenSBI的运行时版本。

在右侧终端中，执行 `b* kern_entry` 后，GDB显示 `Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7`，说明内核入口函数 `kern_entry` 的首地址为0x80200000。

执行 `continue` 后，GDB触发断点并显示 `kern_entry () at kern/init/entry.S:7`，证明OpenSBI已成功将CPU控制权移交到内核入口地址0x80200000。

4. 查看寄存器状态

```
(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>
>
```

我们可以从地址验证、参数传递、执行上下文这三个维度来分析我们看到的寄存器的值。

1. 地址验证 —— 确认启动链路的正确性

- pc (程序计数器): 0x80200000 <kern_entry>

这直接证明CPU已成功进入内核入口函数 `kern_entry`，与OpenSBI约定的内核加载地址完全一致，说明OpenSBI到内核的控制权移交流程无异常。

2. 参数寄存器 —— OpenSBI传递的初始化信息

OpenSBI在移交控制权时，通过 `a0-a7` 寄存器向内核传递硬件与内存的关键参数，这些参数是内核初始化的启动配置：

- a0: 0x0

通常表示当前核心ID（RISC-V多核心系统中，引导核心的ID为0），内核通过此参数确认自己是唯一活跃的引导核心，后续仅需初始化单核心环境。

引导核心是指系统加电或复位后，第一个执行初始化流程、并带动其他核心启动的核心。

- a1: 0x82200000

表示物理内存的结束地址，即DRAM的最高可用地址。

- a2: 0x80200000

表示内核镜像的加载起始地址。

- a6: 0x82200000（与a1一致）

再次传递物理内存结束地址，确保参数的冗余性与可靠性。

- a7: 0x80200000（与a2、pc一致）

再次传递内核加载地址。

3. 栈指针 (sp) 与临时栈 —— 内核初始化前的内存状态

- sp: 0x8001bd80

这是OpenSBI在M态为内核临时分配的栈地址，而非内核自定义栈。内核在 `kern_entry` 中会立即执行栈初始化指令（即之前看到的 `la sp, bootstacktop`），将栈切换为内核自己的独立栈空间。

5. si 单步执行进行验证

```
(gdb) si
0x0000000080200004 in kern_entry ()
    at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) i r sp
sp          0x80203000      0x80203000 <SBI_CONSOL
E_PUTCHAR>
```

可以看到单步执行后的sp变为了 0x80203000，即内核自定义栈顶 `bootstacktop`

这一变化验证了内核入口的第一条关键指令“`la sp, bootstacktop`”的功能 —— 将栈指针从OpenSBI的临时栈切换为内核自己的独立栈。

注：两者在物理地址上完全分离，体现了RISC-V特权级的内存隔离设计，即M态和S态运行在不同的地址空间，避免权限越界和数据干扰。

6. 查看栈内存初始状态

```
(gdb) x/10x 0x8001bd80
0x8001bd80: 0x8001be00 0x00000000 0x8001
be00 0x00000000
0x8001bd90: 0x46444341 0x55534d49 0x0000
0000 0x00000000
0x8001bda0: 0x00000000 0x00000000
(gdb) x/10x $sp
0x80203000 <SBI_CONSOLE_PUTCHAR>: 0x00000001 0
x00000000 0x00000000 0x00000000
0x80203010: 0x00000000 0x00000000 0x0000
0000 0x00000000
0x80203020: 0x00000000 0x00000000
```

执行 `x/10x 0x8001bd80` 后，内存内容呈现明显的随机值特征：

- `0x8001bd80: 0x8001be00 0x00000000 ...`：这些值是OpenSBI在M态执行时的临时数据残留。
- `0x8001bd90: 0x46444341 0x55534d49 ...`：甚至出现ASCII字符的编码（如 `0x46` 对应字符 'F'），是OpenSBI打印日志或操作外设时的残留数据。

综上，**OpenSBI的临时栈**因M态的复杂操作，内存中充满垃圾数据，若内核直接使用此栈，函数调用和局部变量存储会因脏数据而崩溃。

执行 `x/10x $sp` 后，内存内容呈现连续的0值：

- `0x80203000: 0x00000001 0x00000000 ...`（前几个字节可能因初始化指令略有差异，但整体为清零状态）。
- 后续地址（如 `0x80203010`）均为 `0x00000000`，证明内核栈经过显式初始化。

综上，**内核栈**的干净数据确保了函数调用的可靠性，局部变量初始化为0，函数栈帧无残留数据干扰，是内核稳定执行的基础。

7. 反汇编内核入口附近代码

```
(gdb) disassemble kern_entry, +32
Dump of assembler code from 0x80200000 to 0x80200020:
   0x0000000080200000 <kern_entry+0>: auipc sp,0x3
=> 0x0000000080200004 <kern_entry+4>: mv sp,sp
   0x0000000080200008 <kern_entry+8>: j 0x8020
000a <kern_init>
   0x000000008020000a <kern_init+0>: auipc a0,0x3
   0x000000008020000e <kern_init+4>: addi a0,a0,
-2 # 0x80203008
   0x0000000080200012 <kern_init+8>: auipc a2,0x3
   0x0000000080200016 <kern_init+12>: addi a2,a2,
-10 # 0x80203008
   0x000000008020001a <kern_init+16>: addi sp,sp,
-16 # 0x80202ff0
   0x000000008020001c <kern_init+18>: li a1,0
   0x000000008020001e <kern_init+20>: sub a2,a2,
a0
End of assembler dump.
```


从反汇编的结果中，我们可以看到内核入口到初始化函数的执行链路，理解内核如何从汇编入口过渡到C语言初始化逻辑。

kern_entry 的关键指令有：

1. 0x80200000 <kern_entry+0>: auipc sp, 0x3
 - auipc 是RISC-V的基于程序计数器的加法立即数指令，用于计算全局符号的地址（此处为 bootstacktop ）。
 - 功能：初始化内核栈指针，对应之前看到的 la sp, bootstacktop 指令。

它的工作原理是：

$rd = (imm \ll 12) + (pc \& 0xFFFFF000)$

- imm 是指令中的立即数，pc 是当前程序计数器，0xFFFFF000 是取 pc 的高位
- $sp = (0x3 \ll 12) + (0x80200000 \& 0xFFFFF000) = 0x3000 + 0x80200000 = 0x80203000$
- 正好是之前在 kernel.sym 中看到的 bootstacktop 地址。

2. 0x80200004 <kern_entry+4>: mv sp, sp
 - 这是一条空操作指令或调试器显示的简化形式，实际是栈初始化完成后的确认步骤或生成的对齐指令。
3. 0x80200008 <kern_entry+8>: j 0x8020000a <kern_init>
 - j 是无条件跳转指令，直接跳转到 kern_init 函数，正如之前的分析，这是内核从汇编入口到C语言初始化的过渡。

kern_init 是内核的第一个C语言函数，反汇编结果显示了其初始化流程：

1. 栈帧建立

- 0x8020001a <kern_init+16>: addi sp, sp, -16
- 为 kern_init 的局部变量和函数调用建立栈帧，分配16字节栈空间。

2. 参数传递

- li a1, 0 : 将参数 a1 初始化为0。
- sub a2, a2, a0 : 对参数寄存器进行运算，也许是用于计算内存范围，如 a2 是内存结束地址，a0 是核心ID，运算后得到当前核心可管理的内存大小。

8. 继续执行 si

```
(gdb) si
9          tail kern_init
pc         0x80200008      0x80200008 <kern_entry+8>
```

结合反汇编代码可知这次执行的实际上是指令 mv sp, sp，pc指向的指令是 0x80200008 <kern_entry+8>: j 0x8020000a <kern_init>


```
(gdb) si
kern_init () at kern/init/init.c:8
8      memset(edata, 0, end - edata);
(gdb) i r pc
pc      0x8020000a      0x8020000a <kern_init>
```

再次执行si，可以看到pc为0x8020000a，已成功进入C语言函数kern_init。

而memset(edata, 0, end - edata);是内核初始化的内存清零操作——将已初始化数据段(.data)末尾(edata)到未初始化数据段(.bss)末尾(end)的区域(即完整的.bss段)全部置零。

问题回答

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？

1. 地址位于**0x1000**。
2. 主要完成的功能：

我们可以通过反汇编分析0x1000地址指令的具体功能，执行指令x/10i 0x1000，查看0x1000地址开始的10条指令。

```
(gdb) x/10i 0x1000
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:      .insn    2, 0x8000
0x101c:      unimp
```

MROM（机器模式只读存储器）是硬件层面的只读存储介质，其中固化了一段极简的**复位代码**。CPU加电复位后，会从MROM的起始地址（0x1000）读取并执行这段复位代码，这段代码是CPU加电后第一个执行的程序。

- auipc t0, 0x0
 - auipc 将当前pc（0x1000）的高位与立即数0x0拼接，结果存入t0。即 $t0 = (0x1000 \& 0xFFFFF000) + (0x0 \ll 12) = 0x1000$ 。
 - 用于获取当前代码段的基地址，便于后续地址计算。
- addi a1, t0, 32
 - 将t0（0x1000）与立即数32相加，结果存入a1。即 $a1 = 0x1000 + 32 = 0x1020$ 。
 - 计算某一临时数据或配置区域的地址，可能用于存储硬件初始化参数。
- csrr a0, mhartid
 - csrr 读取机器模式核心ID寄存器的值，存入a0。

- 用于识别当前执行的核心ID，如多核心系统中，`mhartid=0` 表示引导核心，用于后续多核心协调。
 - `ld t0, 24(t0)`
 - `ld` 从 `t0+24`（即 `0x1000+24=0x1018`）地址处加载8字节数据，存入 `t0`。
 - 读取预先存储的OpenSBI固件入口地址（QEMU复位代码中，OpenSBI的入口地址被预置在0x1018处）。
 - `jr t0`
 - `jr` 跳转到 `t0` 指向的地址，即OpenSBI固件的入口。
 - 完成硬件复位代码到OpenSBI固件的控制权移交，是启动流程从硬件初始化到固件执行的关键过渡。
-

本实验中重要的知识点及与对应的OS原理中的知识点

- 汇编伪指令与机器指令转换。如.s文件中`la sp, bootstacktop`被汇编器转为 `auipc` 和 `addi`（实际上，低12位为0则无需使用`addi`）机器指令的二进制代码，反汇编可见人类可读的真实汇编指令。
- QEMU模拟器运行内核，就对应OS的虚拟化思想。QEMU 通过软件方式模拟一台完整的RISC-V 计算机，以为自己独占整个资源，实现了对硬件资源的虚拟化。操作系统原理中“虚拟化”强调让多个程序共享底层资源的抽象；而 QEMU 是在实验环境中提供的底层虚拟硬件支持。
- OpenSBI 固件的启动与控制权转移，对应OS的控制权转移和特权级机制。OpenSBI 跳转到 `0x80200000`，对应控制权的转移。OpenSBI 运行在 M 态，内核运行在 S 态，是因为 OpenSBI要有权访问所有的硬件资源。

OS原理中很重要，但在实验中没有对应上的知识点

- OS核心功能模块：
 - 进程管理：操作系统负责创建、调度、终止和同步进程；确保多任务运行时的资源分配与隔离。例如进程。
 - 内存管理：管理物理内存与虚拟内存，负责分配、释放和地址映射。例如页表与TL。
 - 存储与文件管理：提供持久化存储抽象，让用户通过文件接口访问磁盘数据。例如文件系统结构。
- 也可以从设计原则入手：虚拟化、并行、持久化
 - 虚拟化：CPU虚拟化（进程）、内存虚拟化（页表）
 - 并行：线程
 - 持久化：文件系统