ucore Lab 1 实验报告

姓名　　李欣益

2025 年 10 月 9 日

---

1. 实验基本信息与格式说明

- **实验名称：** lab1: 比麻雀更小的麻雀（最小可执行内核）
- **姓名：**李欣益
- **学号：** 2312841

1　练习 1: 理解内核启动中的程序入口操作

1. **指令：`la sp, bootstacktop`**
   - **完成的操作：**我们使用指令 `la` (Load Address) 将汇编代码中定义的标签 `bootstacktop` 的**绝对内存地址**加载到 `sp` 寄存器（栈指针）中。`bootstacktop` 标记的是内核启动栈预留空间（由 `KSTACKSIZE` 定义，通常为 8KB）的**最高地址**, uploaded:lab1/kern/init/entry.S。
   - **目的：**这是执行任何 C 语言代码（例如 `kern_init` ）的**先决条件**。它初始化了内核栈，因为 C 函数调用需要栈来存储返回地址、参数和局部变量。由于 RISC-V 栈是向下增长的，将 `sp` 初始化为最高地址，确保栈有足够的空间可以向下使用。
2. **指令：`tail kern_init`**
   - **完成的操作：**执行一个无条件跳转操作（使用了**尾调用优化**），将 CPU 的程序执行流从汇编代码 `kern_entry` **无条件地转移**到 C 语言的 `kern_init` 函数的入口地址。
   - **目的：**完成**权力交接**。这是将 CPU 的控制权从底层的、依赖汇编的启动阶段，正式移交给高级的 **C 语言初始化阶段**。这样 `kern_init` 函数就可以开始执行 BSS 清零、I/O 初始化等复杂的操作系统逻辑。

2　练习 2: 使用 GDB 验证启动流程

2.1　准备调试环境

- 编译:

  我们使用如下指令:

```
make
```

  确保内核代码已编译完成，并生成了 bin/kernel 可执行文件和 ucore.img 内核镜像。

- 启动 QEMU 调试模式

```
make debug
```

这会启动 QEMU 模拟器，并带上 -s -S 参数。- -S: 告诉 QEMU 暂停 CPU，等待 GDB 连接。- -s (或 -gdb tcp::1234): 开启 GDB 远程调试端口（默认 1234）。

- 启动 GDB 客户端

我们开启新的 cmd 窗口使用：

```
make gdb
```

GDB 客户端启动并连接到 QEMU 模拟器。由于 QEMU 处于暂停状态，CPU 此时应该停留在启动的第一条指令之前。

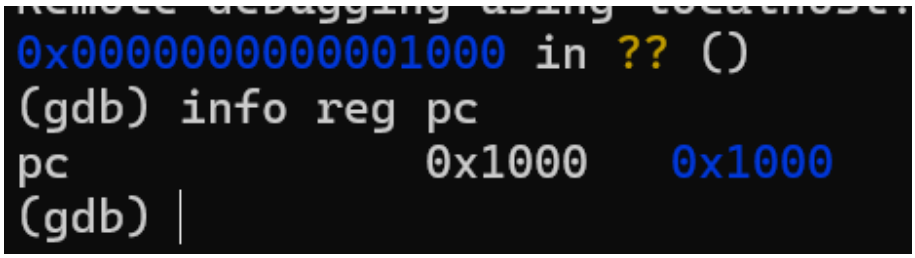
2.2 观察 SBI 固件启动流程

- 检查起始地址

我们使用指令：

```
info reg pc
```

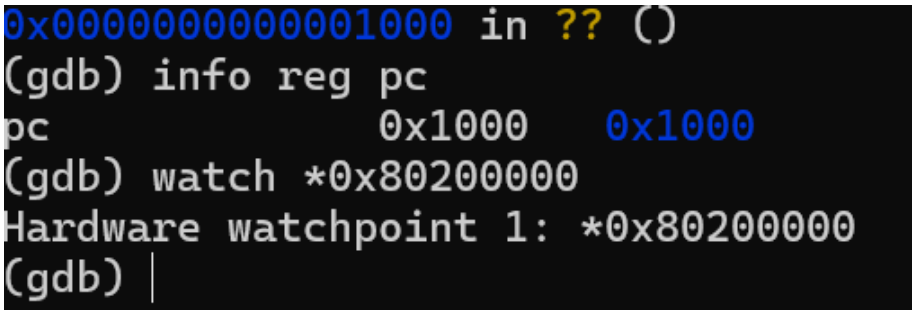
检查程序计数器（PC）的值。



PC 位于 RISC-V 的复位地址 0x1000, 即为加电后 CPU 执行的第一条指令所在的地址。这是 OpenSBI 代码开始执行的地方。

- 设置内核加载观察点

```
watch *0x80200000
```

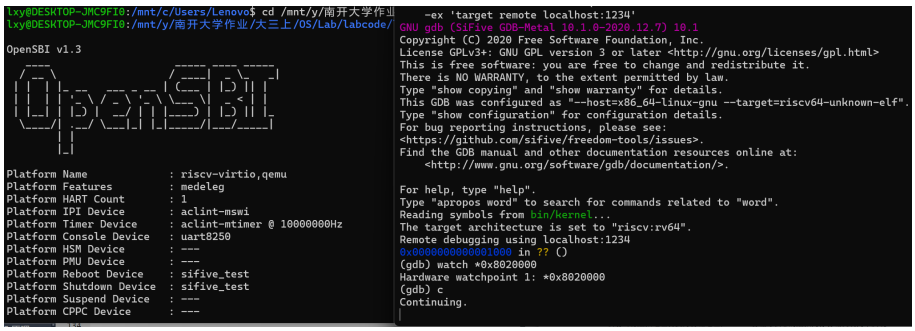


在内核的加载地址设置一个观察点 (Watchpoint)。这样 GDB 会在内存 0x80200000 处的内容发生变化时暂停。这能捕获 OpenSBI 将内核镜像写入该地址的瞬间。

- 继续执行 OpenSBI 固件

```
c
```

GDB 会暂停，报告观察点被触发。这捕获了 OpenSBI 将内核代码写入内存的瞬间。



OpenSBI 固件运行得非常快，它在连接 GDB 或设置观察点之前，可能已经将内核代码写入了 0x80200000。因此，GDB 错过了那个写入瞬间

左图中 make debug 输出了 OpenSBI 报告（OpenSBI v1.3 Banner 及后续的配置信息），说明当我在 GDB 中输入 c 后，OpenSBI 代码开始运行

2.3 观察内核入口跳转

- 移除观察点

```
del 1
```

我们移除刚才设置的观察点（Watchpoint），避免干扰后续执行。

- 设置内核入口断点

```
b *0x80200000
```

在 ucore 内核入口地址设置一个断点 (Breakpoint)。

- 继续执行

```
c
```

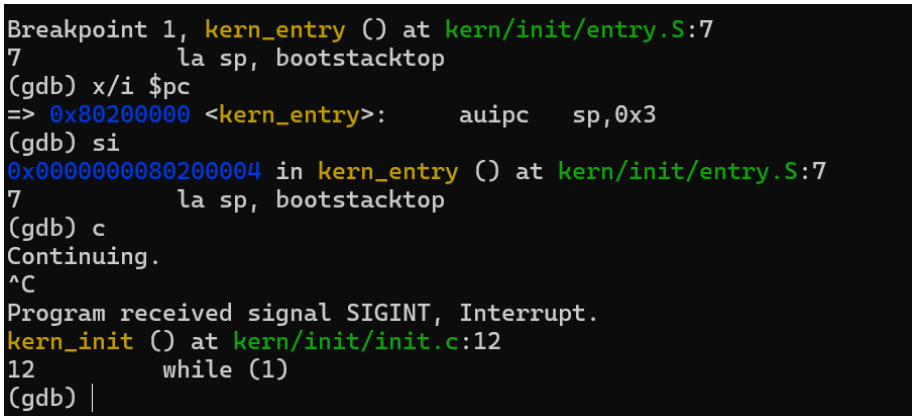
OpenSBI 继续执行，完成权限切换等任务，最终跳转到内核入口，触发断点。

- 抵达内核

```
si

```

GDB 暂停在地址 0x80200000。单步执行后，执行的是 entry.S 中的第一条指令。



我们发现，断点命中后，使用 si 单步执行 la sp, bootstacktop，这即为 entry.S 中的第一条指令

综上，我们回答练习 2 的问题：

- Q: RISC-V 硬件加电后最初执行的几条指令位于什么地址?

  A: 最初执行的地址：0x1000。

  执行的代码归属：OpenSBI 固件代码.

- 它们主要完成了哪些功能?

  - 底层硬件初始化：OpenSBI（M-Mode）进行 CPU 核心、中断控制器等最基础的硬件设置。
  - 内核加载/放置：确认内核镜像（ucore.img）被放置到内存的 0x80200000 地址。
  - 权限交接与跳转：设置 CPU 特权模式从 M-Mode 降级为 S-Mode（监督者模式），并最终跳转到 0x80200000，将控制权移交给 ucore 内核

  。

3 实验总结与原理分析

3.1 实验重要知识点和 OS 原理对照

表 1: 重要知识点与 OS 原理对照

实验中的知识点	对应的 OS 原理知识点	含义、关系、差异等方面的理解
`kernel.ld` (链接脚本)	静态内存布局	**含义:** 链接脚本是编译阶段就确定的内核在内存中的结构划分，如 `.text`、`.data`、`.bss` 等区域。**关系:** 这是操作系统实现地址空间划分和分区管理的起点。
`ecall` / SBI 接口	特权级分离与系统调用	**含义:** `ecall` 是从低权限的 S-Mode (内核) 切换到高权限的 M-Mode (OpenSBI 固件) 的指令。**关系:** OpenSBI 利用 SBI 接口充当了 S 层和 M 层对话的桥梁，是实现 I/O 和硬件访问的唯一合法途径。
I/O 分层 (`stdio.c` → `console.c`)	I/O 驱动抽象	**含义:** 将高级格式化逻辑 (`stdio.c`) 与底层硬件通信 (`console.c`) 解耦。**差异:** 这种分层提高了内核的可移植性，当底层硬件变化时，只需修改 `console.c`，而无需改动 `stdio.c`。
`la sp, bootstacktop`	栈初始化	**含义:** 这是 CPU 获得控制权后，为 C 语言环境设置 `sp` 寄存器（栈指针）的操作。**关系:** 这是执行任何高级代码（如 `kern_init`）的先决条件。
BSS 段清零 (`memset` → `edata`, `end`)	运行时数据初始化	**含义:** 在 `kern_init` 中手动将所有未初始化全局变量（`.bss` 段）清零。**关系:** 这是遵循 C 语言规范（未初始化变量默认为 0）的关键步骤，确保系统数据一致性。

3.2 OS 原理中重要但实验中未涉及的知识点

- **虚拟内存管理:** 实验中虽然定义了页大小 (`PGSIZE=4096`)，但未实现页表的建立、虚拟地址到物理地址的映射、TLB（Translation Lookaside Buffer）管理等核心机制。
- **进程与线程管理:** 本实验代码只有一个执行流 (`kern_init`)，没有涉及到 PCB（进程控制块）、进程/线程创建、调度算法、上下文切换和同步互斥等核心并发管理概念。

4 gitee 链接

Gitee 仓库：`https://gitee.com/dashboard/projects`