南開大學
Nankai University

计算机学院

操作系统实验报告

Lab1

姓名：蔡子轩

学号：2310984

专业：计算机科学与技术

2025 年 10 月 10 日

目录

1 实验目的

本实验旨在在 QEMU 模拟的 64 位 RISC-V 平台上构建并启动一个最小可执行内核，弄清从固件到内核 C 入口的完整启动路径及其所需的构建要素（内存布局、链接与镜像生成）。

- 理解启动链路中各角色的分工与衔接：OpenSBI（前置固件/引导）→ 内核入口 `kern_entry` → C 层 `kern_init`。
- 学会用链接脚本描述内核镜像的内存布局（如 `.text`/`.rodata`/`.data`/`.bss`、启动栈等）。
- 掌握使用 RISC-V 交叉工具链进行编译与链接，生成可在 QEMU 上运行的内核镜像。
- 会以 OpenSBI 作为“bootloader”加载并启动内核，在 QEMU 中完成启动验证。
- 能调用 SBI 控制台等服务实现格式化打印，为后续调试与功能扩展打基础。

2 理解内核启动中的程序入口操作

2.1 背景：从 OpenSBI 到 kern_entry

QEMU 复位后，CPU 从 `0x1000` 进入 MROM，再跳到 `0x80000000` 执行 OpenSBI。OpenSBI 完成最小硬件环境初始化后，将内核镜像加载到 `0x80200000` 并跳转过去。**链接脚本**把内核入口符号设为 `kern_entry`，因此控制权到达我方代码的第一条指令就是 `kern/init/entry.S` 中的 `kern_entry`，因此我们需要对 `kern/init/entry.S` 做一定的了解。

2.2 相关代码（kern/init/entry.S）

```
#include <mmu.h>
#include <memlayout.h>
    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop
    tail kern_init
.section .data
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

2.3 逐行解释

- `#include <mmu.h>`, `#include <memlayout.h>`：引入常量/宏（如 `PGSHIFT`、`KSTACKSIZE`），用于对齐与大小。

- `.section .text,"ax",%progbits`：其原型为 `.section name,"flags",%type`，表示切换到（或创建）名为 name 的节（section），并同时设置其属性（flags）与类型（type）。其中 `.text` 为节名，按惯例用于代码段，等同于“将后续指令放入 `.text` 节”。随后 `"a"` 表示该节在装载时需要映射进内存（alloc），`"x"` 表示该节内容可执行（execute），`%progbits` 指节类型为 SHT_PROGBITS（内容真实存在于目标文件中，区别于 `%nobits` 仅占内存不占文件）。总体而言，`.section .text,"ax",%progbits` 的含义是：切到名为 `.text` 的节；该节会被装载进内存（a）、可执行（x），且其内容存在于文件中（progbits）。简而言之，后续指令都属于可执行的代码段。

- `.globl kern_entry`：`.globl`（同 `.global`）是 GNU as 伪指令，用于将符号声明为全局符号（STB_GLOBAL），使其在链接阶段对其他目标文件可见；它本身不生成机器指令、也不分配存储。这里把标签 `kern_entry` 声明为全局后，链接脚本通常通过 `ENTRY(kern_entry)`（或链接器参数 `-e kern_entry`）将其写入 ELF 头的 `e_entry` 字段，作为内核镜像的程序入口。处理器/引导程序把控制权转交到 `kern_entry` 后，该位置的代码完成早期准备（如 `la sp, bootstacktop` 设置内核启动栈），随后“无返回”跳转到 C 层入口 `kern_init` 开始更高层初始化。

- `kern_entry:`：内核拿到控制权后执行的第一条指令位置。

- `la sp, bootstacktop`：把**内核栈的高地址边界**（预留区上沿）写入 `sp`，为 C 代码提供干净、对齐的早期内核栈（栈从高往低生长）。

- `tail kern_init`：**无返回**地跳到 C 入口 `kern_init`，完成控制权移交。

- `.section .data`：切换到 ELF 的 `.data` 节（已初始化、可写、装载时会映射到内存）。

- `.align PGSHIFT`：`.align` 是汇编伪指令，用于将**当前位置**对齐到指定边界；在 GNU as 的 ELF 目标上，`.align n` 表示按 2^n 字节对齐（等价于 `.p2align n`），需要时会自动用填充字节补齐。`PGSHIFT` 是头文件中的宏，表示**页大小的幂指数**。因此，`.align PGSHIFT` 的含义是“将当前位置对齐到 2^{PGSHIFT} 字节边界（常见 `PGSHIFT=12`，即 4 KiB 页边界）”，保证随后预留的启动栈从整页开始，便于内存管理与保护。

- `.global bootstack`：将符号 `bootstack` 声明为全局，使其在链接时对其他目标文件可见；与后面的标签 `bootstack:` 一起，表示早期内核栈在.data 节中的起始地址可被外部引用。

- `bootstack:`：在当前 `.data` 节内定义标签 `bootstack`，标记预留栈区的起始地址（低地址端）。由于前面使用了 `.align PGSHIFT`，该起点通常是整页对齐的。

- `.space KSTACKSIZE`：GAS 的数据伪指令，用于在当前节中顺序生成指定数量的字节。.space size[, fill] 的含义是“生成 `size` 个字节，并用可选的 `fill` 值填充（默认 0）”；它与 `.skip` 等价。这里会在 `.data`中连续放入 `KSTACKSIZE` 个字节，因此这块内存被静态编入镜像文件，装载时会映射到内存，作为早期内核的启动栈存储区。

- `.global bootstacktop`：将符号 `bootstacktop` 声明为全局，使其在链接阶段对其他目标文件/调试器/链接脚本可见，便于引用这块栈的上沿地址。

- `bootstacktop:`：在当前位置再定义标签 `bootstacktop`。由于它紧随 `.space KSTACKSIZE` 之后，故有 `bootstacktop - bootstack == KSTACKSIZE`；它正好是这段预留栈区的高地址边界（栈顶上沿）。随后用 `la sp, bootstacktop` 将 `sp` 指向这里，栈将自高地址向低地址生长。

2.4 两条关键指令的操作和目的

2.4.1 la sp, bootstacktop

操作：把符号 bootstacktop 对应的“地址数值”写入寄存器 sp。

目的：建立一个**对齐**、**可用**、**向低地址生长**的早期内核栈，让随后的 C 代码（kern_init 以及其调用链）能安全进行入栈/出栈、保存现场、创建栈帧等。选择“bootstacktop（高地址边界）作为初始 sp”是为了第一次压栈时向低地址移动，恰好落入我们在 .data 段**预留**的这块空间中。

2.4.2 tail kern_init

操作：不留下返回地址，把控制权直接交给 C 入口 kern_init。

目的：启动桩的使命是“把机器状态拉到能跑 C 的地方并**完成交接**”，它**不应**再返回。无返回跳转使控制流更干净，避免无意义的返回路径与开销，也不污染返回地址寄存器。

2.5 与启动执行流的对应关系

1. OpenSBI 跳转至 kern_entry；
2. la sp, bootstacktop：就位内核栈；
3. tail kern_init：无返回地进入 C 入口；
4. kern_init 内调用 cprintf 等，打印启动成功信息。

2.6 对内存栈理解的示意

```
0x....4000  ← bootstacktop  （高地址边界；空栈时 sp 放在这里）
    ↓  压栈：sp 先减再写，数据向低地址“生长”
... ...
0x....2000  ← bootstack     （低地址边界；预留区间下沿）
```

栈向低地址生长：**push => sp 变小**；**pop => sp 变大**。

2.7 什么是 kern_init ？为什么要有“早期内核栈”？

kern_init 是内核在 C 语言层的第一个入口函数，作用等价于“内核态的 main()”。在本实验的最小可执行内核中，kern_init 主要用于验证启动通路是否畅通（例如调用 cprintf 打印启动信息）。在后续实验中，它将逐步承担并下发更多初始化任务（内存管理、异常中断、调度器、驱动等），是内核所有子系统启动的起点。

为了能在进入 kern_init 后立刻进行 C 函数调用与嵌套（保存返回地址、局部变量、参数传递等），必须在进入 C 代码之前先建立一块可用的栈空间。为此，entry.S 在 .data 段静态预留了一块按页对齐的连续内存作为“早期内核栈”，并通过 la sp, bootstacktop 将 sp 指向其高地址边界（栈向低地址生长）。这块早期栈无需分页和动态分配即可使用，保证了 C 层初始化的安全与可预测性；在后续建立每 CPU/线程专用栈后，该引导期栈即可退出历史舞台或作为保护备用。

2.8 总结

`la sp, bootstacktop` 用来**就位内核栈**（高地址上沿，向低地址生长），`tail kern_init` 用来**无返回地移交控制权**到 C 入口；两者合起来完成“从汇编启动桩进入 C 世界”的最后一跳。

3 使用 GDB 验证启动流程

3.1 实验目标与层次

目标：用 GDB 远程调试 QEMU，**从加电复位（0x1000）跟到内核第一条指令（0x80200000）**，并用现场证据解释各阶段作用。

分层视角：

- **MROM（0x1000）**：复位向量，最早执行的固件指令（上电后第一条）。
- **OpenSBI（0x80000000, M-mode）**：平台早期初始化、按 `mhartid` 分流从核、准备移交控制权。
- **内核入口(0x80200000, S-mode)**:`kern_entry`;`la sp, bootstacktop` 设早期栈,`tail kern_init` 无返回跳入 C 入口。

3.2 实验命令与详细流程

3.2.1 进入 GDB

在另一个终端先 `make debug`，本终端执行 `make gdb`。第一次进 GDB 会有分页提示：

```
1 ——Type <RET> for more, q to quit, c to continue without paging——
```

这里我直接按 `c`，关掉这一次的分页。随后 GDB 输出显示：已经加载内核符号（Reading symbols from bin/kernel...）、设置了架构(`riscv:rv64`)、并连上 QEMU(`Remote debugging using localhost:1234`)。更关键的是，**PC 此刻正好在 0x1000**，也就是复位向量 / MROM 的起点。

```
wsy@sparewheel:~$ cd ~/oslab/labcode/labcode/lab1
wsy@sparewheel:~/oslab/labcode/labcode/lab1$ make -j
make: Nothing to be done for 'TARGETS'.
wsy@sparewheel:~/oslab/labcode/labcode/lab1$ make debug
qemu-system-riscv64 \
        -machine virt \
        -smp 1 \
        -nographic \
        -bios default \
        -device loader,file=bin/ucore.img,addr=0x80200000 \
        -s -S

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)
```

make debug

```
wsy@sparewheel:~$ cd ~/oslab/labcode/labcode/lab1
wsy@sparewheel:~/oslab/labcode/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-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 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 redistribute 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-linux-gnu --target=riscv64-unknown-elf".
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 <RET> for more, q to quit, c to continue without paging--c
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
0x0000000000001000 in ?? ()
```

make gdb

3.2.2 友好化显示

我们输入以下命令行，进行友好化显示：

```
(gdb) set pagination off
(gdb) set disassemble-next-line on
(gdb) display /i $pc
1: x/i $pc
=> 0x1000:      auipc   t0,0x0
```

三条设置的效果是：输出不再被分页打断；每次停住，GDB 自动把“下一条将执行的指令”反汇编给我看；并且总是显示当前 `$pc` 所在的那条指令。现在一眼就能看出，**0x1000 的第一条是** `auipc t0,0x0`。

3.2.3 从复位向量到 OpenSBI

```
(gdb) x/12i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   ...
(gdb) advance *0x1010
0x0000000000001010 in ?? ()
=> 0x0000000000001010:  67 80 02 00     jr      t0
1: x/i $pc
=> 0x1010:      jr      t0
(gdb) si
0x0000000080000000 in ?? ()
=> 0x0000000080000000:  73 28 40 f1     csrr    a6,mhartid
1: x/i $pc
=> 0x80000000:  csrr    a6,mhartid
```

这一小步就是把“第一跳”拍实：MROM 在 0x1000 附近做了几件事——读核 ID（`csrr mhartid`）、从内部表里取下一阶段入口（`ld t0,...`），最后 `jr t0` 跳过去。我用 `advance *0x1010` 停在 `jr t0` 上，再 `si` 单步，**PC 立刻从 0x1010 跳到了 0x80000000**，这就是 OpenSBI 的入口。

3.2.4 “三站式”断点：复位/固件/内核

```
(gdb) delete breakpoints
(gdb) hbreak *0x1000
Hardware assisted breakpoint 1 at 0x1000
(gdb) hbreak *0x80000000
Hardware assisted breakpoint 2 at 0x80000000
(gdb) tbreak *0x80200000
Temporary breakpoint 3 at 0x80200000: file kern/init/entry.S, line 7.
```

这里清掉历史断点后，分别在复位点(0x1000)、OpenSBI 起点(0x80000000)打“硬件断点”(ROM/固件区域不能插软件断点)，再在内核入口（0x80200000）打一次性断点，方便后面“一路跑到第三站就停”。

3.2.5 软复位并放行

```
(gdb) monitor system_reset
(gdb) c
Continuing.
Breakpoint 2, 0x0000000080000000 in ?? ()
=> 0x80000000:  csrr    a6,mhartid
```

`monitor system_reset` 让虚拟机回到“刚上电”的状态；`c` 放行后，**经常先命中 0x80000000**。这是因为 GDB 和 QEMU 的同步不是每条指令都握手，MROM 的那几条非常快，`c` 之后它一口气做完第一跳，GDB 抢回控制权时就已经在 OpenSBI 入口了。这个现象不影响我们在上一小节用单步把“第一跳”确认清楚。

3.2.6 查看 OpenSBI 开头

```
(gdb) x/12i $pc
=> 0x80000000:  csrr a6,mhartid
   0x80000004:  bgtz a6,0x80000108
   ...
```

这一步用 x/12i $pc 在 0x80000000 处连续反汇编 12 条，我们可以得知，反馈输出基本套路是：先读 `mhartid`，如果不是 0 号核（`a6>0`），就分流去等待（一般是 `wfi` 循环）；**只有 hart0 继续主初始化**并最终把控制权交给内核。

3.2.7 继续第二跳：进入内核

```
(gdb) c
Continuing.

Temporary breakpoint 3, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
=> 0x80200000 <kern_entry>: auipc sp,0x3
   0x80200004 <kern_entry+4>: mv sp,sp
```

放行后命中第三站：PC 到了 0x80200000，也就是内核的第一条指令 kern_entry。源码对应的第一件事很直观：la sp, bootstacktop——把早期内核的栈顶先立起来。

3.2.8 收尾：设好栈，并以 `tail` 进入 C 入口

```
(gdb) x/6i $pc
=> 0x80200000 <kern_entry>: auipc sp,0x3
   0x80200004 <kern_entry+4>: mv sp,sp
```

```
   0x80200008 <kern_entry+8>: j 0x8020000a <kern_init>
(gdb) si
... 仍在 kern_entry, 第二条显示为 mv sp,sp
(gdb) si
9           tail kern_init
=> 0x80200008 <kern_entry+8>: j 0x8020000a <kern_init>
```

这里的两次 `si` 实际上把 `la sp, bootstacktop` 展开的两条机器指令（auipc sp,imm20 + addi sp,sp,imm12)跑完了,SP 被精确设成了 bootstacktop,早期栈就绪;紧接着的 `j 0x8020000a <kern_init>` 等价于 `jal x0,kern_init`，也就是所谓的 *tail* 调用：**不写返回地址、无返回跳转**，控制权直接交给 C 入口 `kern_init`，不会再回到 `kern_entry`。

到此为止，两次关键跳转——**0x1000 → 0x80000000 → 0x80200000**——全部有据可查。

3.3 问题回答：最初指令在哪，做了什么？

位置： QEMU/virt 平台上电后，CPU **从 `0x1000`（MROM 复位向量）开始执行**。

我们在 GDB 的现场证据：

```
(gdb) x/12i 0x1000
=> 0x1000: auipc t0,0x0
   0x1004: addi  a1,t0,32
   0x1008: csrr  a0,mhartid
   0x100c: ld    t0,24(t0)
   0x1010: jr    t0
   ...
(gdb) advance *0x1010
=> 0x1010: jr t0
(gdb) si
=> 0x80000000: csrr a6,mhartid    ; OpenSBI 起点
```

功能解释（逐条）：

- `auipc t0,0`：用当前 PC 的高 20 位给 `t0` 做**位置无关基址**，后续可在 ROM 内取数据/表项;
- `addi a1,t0,32`：`a1` 指向“基址 +32”的一块信息（向下一阶段**传参指针**）;
- `csrr a0,mhartid`：读当前 **hart ID** 到 `a0`，按调用约定传给下一阶段;
- `ld t0,24(t0)`：从“基址 +24”**读取下一阶段入口地址**（本机为 `0x80000000`）;
- `jr t0`：**跳转**到该入口，转入 **OpenSBI** 固件执行。

小结： MROM 在 `0x1000` 附近完成“取基址 → 准备参数（*a0=hartid, a1=* 指针）→ 从表项取下一阶段入口 → 跳转到 *0x80000000*”的最小化引导，然后由 OpenSBI 完成平台初始化并**第二跳**到 `0x80200000`（`kern_entry`），内核第一条设定早期栈并 `tail kern_init` 进入 C 入口。