实验报告：Lab1 —— RISC-V 引导与裸机启动

一、实验概述

1. 实验目标

通过参考 xv6 的裸机引导流程，实现 `_entry → start() → UART` 的最小引导链路，掌握 RISC-V 启动阶段的栈初始化、BSS 段清零、串口驱动与多核栈隔离关键机制。

2. 完成情况

- `entry.S` 完成 per-hart 栈指针设置、BSS 清零与跳转逻辑。
- `kernel.ld`、`start.c`、`uart.c`、`print.c` 等核心文件补全，并为 printf 系列加锁。
- QEMU virt 平台多次运行 `make qemu` 均稳定输出 “Hello OS”，无未完成任务。

3. 开发环境

- 硬件：x86_64 主机（2 核）
- 操作系统：Ubuntu 24.04 LTS
- 交叉工具链：riscv64-unknown-elf-gcc 12.2.0
- 模拟器：QEMU System riscv64 8.2.2
- 调试器：gdb-multiarch 15.0.50.20240403

4. 实验目录结构

```
###  代码组织结构
whu-oslab-lab1
├── include
│   ├── uart.h
│   ├── lib
│   │   ├── print.h
│   │   └── lock.h
│   ├── proc
│   │   ├── cpu.h
│   │   └── proc.h
│   ├── common.h
│   ├── memlayout.h
│   └── riscv.h
├── kernel
│   ├── boot
│   │   ├── main.c
│   │   ├── start.c
│   │   ├── entry.S
│   │   └── Makefile
│   ├── dev
│   │   ├── uart.c
│   │   └── Makefile
│   ├── lib
│   │   ├── print.c
```

```
│   │   ├── spinlock.c
│   │   └── Makefile
│   ├── proc
│   │   ├── pro.c
│   │   └── Makefile
│   ├── Makefile
│   └── kernel.ld
├── picture
│   └── *.png
├── Makefile
├── common.mk
├── README.md
└── Report.md
```

二、技术设计

1. 系统架构设计

整体流程分为“加载 → 汇编引导 → C 初始化 → 驱动输出”四个阶段，模块与 xv6 对应部分一一映射，方便后续扩展。

```
QEMU loader  →  _entry(entry.S)  →  start()  →  print_init()/puts()
      |                |               |               |
  加载到0x80000000   栈和BSS就绪      主核判定       UART 驱动输出
```

- `boot`：设置 per-hart 栈、清零 BSS，并跳转到 C 入口。
- `lib`：提供 print、自旋锁、assert，保证串口输出线程安全。
- `dev`：实现 16550A UART 驱动，统一寄存器访问接口。
- `proc`：定义 `struct cpu` 与 `mycpuid()`，为多核调度预留接口。

与 xv6 对比：本实验精简了中断、调度和分页，但保留 per-hart 栈与锁语义，后续扩展无需推翻现有设计。

2. 关键数据结构

```c
struct cpu {
    int id;
    int started;
};

typedef struct spinlock {
    int locked;
    char *name;
    int cpuid;
} spinlock_t;

__attribute__((aligned(16))) uint8 CPU_stack[4096 * NCPU];
extern int panicked;
```

- `struct cpu`：通过 `tp` 寄存器获取 hartid，便于锁归属检查。
- `spinlock_t`：记录锁状态与持有者，panic 时可打印锁名，定位死锁。
- `CPU_stack` / `panicked`：置于 BSS 段并由 `_entry` 清零，确保所有核状态一致。

3. 核心算法与流程

1. `_entry` 读取 `mhartid`，为每个 hart 分配 4KB 栈，并以 8 字节步长清零 `[edata,end)`。
2. 跳转到 `start()` 后仅允许 hart0 初始化 UART、打印 “Hello OS”，其余核进入 `wfi`。
3. `print.c` 的 puts/printf 通过自旋锁保护；`uart_putc_sync` 轮询 LSR，保证裸机串口输出可靠。
4. 若后续加入中断/调度，只需在 `start()` 中扩展 S 模式切换与 trap 向量，无需重写引导流程。

三、实现细节与关键代码

1. 关键函数

`entry.S` —— 多核栈与 BSS 清零：

```
la   sp, CPU_stack
li   a0, 4096
csrr a1, mhartid
addi a1, a1, 1
mul a0, a0, a1
add sp, sp, a0
la   a0, edata
la   a1, end
1: bge a0, a1, 2f
   sd   zero, 0(a0)
   addi a0, a0, 8
   j    1b
2: call start
```

`start.c` —— 仅主核初始化 UART 并输出：

```
void start(void) {
    unsigned long hartid;
    asm volatile("csrr %0, mhartid" : "=r"(hartid));
    if (hartid == 0) {
        print_init();
        puts("Hello OS");
    }
    while (1) asm volatile("wfi");
}
```

`print.c` / `uart.c` —— 串口输出与锁保护：

```
void puts(const char *s) {
  if (!s) return;
```

```
  spinlock_acquire(&print_lk);
  while (*s) uart_putc_sync(*s++);
  spinlock_release(&print_lk);
}

void uart_putc_sync(int c) {
  push_off();
  while (panicked);
  while ((ReadReg(LSR) & LSR_TX_IDLE) == 0);
  WriteReg(THR, c);
  pop_off();
}
```

2. 难点突破

- **工具链缺失**：默认使用 `riscv64-linux-gnu-gcc`，改为安装 `gcc-riscv64-unknown-elf` 并统一前缀。
- **汇编扩展名**：`entry.s` 不经过预处理导致符号缺失，更名为 `.S` 并在 `kernel.ld` 导出 `edata/end`。
- **字符串 relocation 溢出**：`"Hello OS"` 距离 PC 过远，调整链接脚本让 `.rodata` 紧挨 `.text`。
- **多核串口乱序**：移除 `hartid==0` 后字符交织，通过自旋锁保护并限制主核输出，恢复稳定性。

3. 源码理解与思考题

1. 源码理解总结

- **启动流程**：QEMU 加载 kernel → `_entry` 设置栈 → 清零 BSS → `start()` → 初始化 UART → `puts("Hello OS")`。
- **内核模块划分**：boot 负责硬件初始化，lib 负责基本功能，proc 提供 CPU 抽象。

2. 思考题解答

- **启动栈**：按 4KB 估算普通函数深度，可在栈底放置魔数检测溢出。
- **BSS 清零**：若省略，`panicked`、`print_lk` 等全局变量将含随机值；只有引导固件保证清零时才可跳过。
- **与 xv6 对比**：缺少中断、分页与调度，但依旧保留 per-hart 栈、自旋锁和 panic 逻辑，有利于扩展。
- **错误处理**：UART 失败无法返回，只能死循环并通过 LED/蜂鸣器输出错误码，保证最小可观测性。

四、测试与验证

1. 功能测试

基本启动

```
$ make qemu
Hello OS
```

```
xsy@XSY:~/whu-oslab-lab1$ make qemu
make build --directory=proc/
make[2]: Entering directory '/home/xsy/whu-oslab-lab1/kernel/proc'
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2
-MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-st
ack-protector -fno-pie -no-pie -I ../../include -c proc.c
make[2]: Leaving directory '/home/xsy/whu-oslab-lab1/kernel/proc'
ls: cannot access './*/*/*.o': No such file or directory
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel.ld -o ../kernel-qemu ./boo
t/entry.o ./boot/main.o ./boot/start.o ./dev/uart.o ./lib/print.o ./lib/spinloc
k.o ./proc/proc.o
riscv64-linux-gnu-ld: warning: ../kernel-qemu has a LOAD segment with RWX permi
ssions
make[1]: Leaving directory '/home/xsy/whu-oslab-lab1/kernel'
qemu-system-riscv64 -machine virt -bios none -kernel kernel-qemu  -m 128M -smp
2 -nographic
Hello OS
Hello OS
```

实际输出与预期一致，串口没有出现乱码。

多核验证

移除 `hartid==0` 判断，每个核都会输出 “Hello OS”，次数与 `-smp` 参数一致，验证 per-hart 栈正常工作。

2. 边界与异常测试

- **BSS 清零实验**：注释清零循环后，`panicked` 有时为 1，串口停止输出，证明清零步骤必要。
- **UART 忙等待**：在 `uart_putc_sync` 中临时增加延迟，确认锁不会被打断，输出仍保持顺序。
- **panic 场景**：手动调用 `panic("test")`，串口输出 `panic: test` 并进入死循环，异常路径可用。

3. 调试与截图

- 调试流程：`make qemu-gdb` + `gdb-multiarch kernel.elf`，`target remote :1234` 后可单步跟踪 `_entry`。
- 截图：`picture/test.png` 展示基本输出；`picture/QQ_1758960389260.png` 展示移除 hart 限制后的多次打印。

五、问题与总结

1. 遇到的问题

1. **entry.s 无法解析符号**
    - 现象：链接时报 `undefined reference to edata`。

- 原因：`.s` 文件缺少 C 预处理阶段。
- 解决：改为 `entry.S` 并在 linker script 中导出符号。
- 预防：约定所有需要包含头文件的汇编文件均使用 `.S`。

2. **字符串 relocation 溢出**

- 现象：编译器提示 `relocation truncated to fit`。
- 原因：字符串常量距离 PC 超过 4KB。
- 解决：将 `.rodata` 紧跟 `.text`，让常量处于可寻址范围。
- 预防：保持段布局紧凑，必要时拆分文本。

2. 实验收获

- 掌握 RISC-V 裸机引导链路、链接脚本与 BSS 管理。
- 理解 16550A UART 配置及多核串口同步方式。
- 熟悉 QEMU + gdb-multiarch 联合调试，能逐条验证指令执行。

3. 改进方向

- 增加 LED/蜂鸣器等最小错误指示，提升裸机可观测性。
- 在 `print.c` 中实现 `printf`/格式化输出并补充单元测试。
- 预留 trap 向量与分页框架，为后续实验快速扩展调度与内存管理。