

Operation System Lab 1

成员：禹相祐 (2312900) 查科言 (2312189) 董丰瑞 (2311973) [TOC]

Operation System Lab 1

一、练习一

二、练习二

(1) Running ucore

(2) GDB调试

① CPU从 0x1000 进入 MROM

② 跳转到 0x80000000

③ OpenSBI初始化并加载内核到 0x80200000 ,进入 kern_entry

④ 调用 kern_init(), 并输出信息

三、实验&原理对应点

四、本实验未覆盖的OS重要原理

一、练习一

在操作系统内核启动流程中，`kern/init/entry.S` 是内核的入口汇编代码，负责完成从底层硬件到内核C代码执行的过渡

(1) `la sp,bootstacktop` 完成了什么操作，目的是什么？

- 作用：初始化内核栈

- 将 `bootstacktop` 的地址加载到 *RISC-V* 的栈指针寄存器 `sp` (指向栈顶部) 中

- 目的：

- 在内核启动的早期即汇编阶段，首要任务是为后续的C代码执行准备运行环境，栈初始化是基础的一步，C语言的函数调用、局部变量的存储，返回地址保存都依赖栈来实现。**内核启动时，`sp`寄存器的值不确定，直接运行C代码会导致栈访问错误。**

(2) `tail kern_init` 完成了什么操作，目的是什么？

- 作用：无返回的跳转到 `kern_init` 函数

- `tail` 是 *RISC-V* 汇编中的伪指令 (实际会被编译为 `jr ra` 等指令)

- `tail` 与普通跳转指令 (如 `j`) 的区别：`tail` 会优化函数调用的栈帧，可以释放调用者的栈空间

- 本质是将 `kern_init` 的地址写入程序计数器 `pc`，且不保留当前函数的返回信息。

- 目的：

- 完成阶段的初始化工作后，将执行流从汇编代码切换到 C 语言代码，进入内核的正式初始化流程，如初始化内存管理、设备、进程调度等。

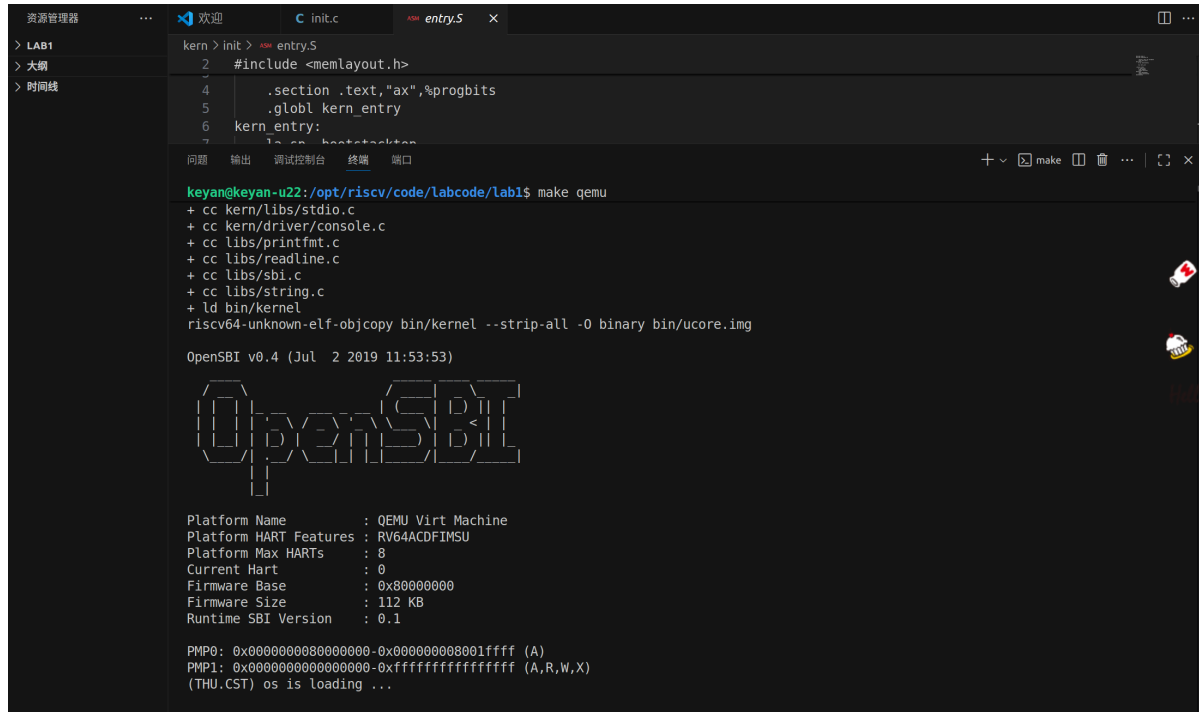
二、练习二

(1) Running ucore

根据知道书中的内容可以学习到：最小可执行内核的完整启动流程为：

- 1 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 → 结束

我们首先在根目录下执行命令 `make qemu`，可以看到



```
kern > init > asm entry.S
2  #include <memlayout.h>
4  .section .text,"ax",%progbits
5  .globl kern_entry
6  kern_entry:
7  .li 0, bootstacktop

keyan@keyan-u22:/opt/riscv/code/labcode/lab1$ make qemu
+ cc kern/libs/stdio.c
+ cc kern/driver/console.c
+ cc libs/printfmt.c
+ cc libs/readline.c
+ cc libs/sbi.c
+ cc libs/string.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

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: 0x0000000000000000-0x0000000000001fff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

输出一行 `(THU.CST) os is loading`，进入了死循环，和我们的C代码相符

(2) GDB调试

我们接下来逐步验证：

我们使用 `tmux` 工具同时查看两个终端，一个用来运行 `qemu`，一个用来运行 `GDB`

在左侧运行指令 `make debug`，右侧运行 `make gdb`，执行完如下：

```
keyan@keyan-u22:/opt/riscv/code/labcode/lab1$
keyan@keyan-u22:/opt/riscv/code/labcode/lab1$ make debug

keyan@keyan-u22:/opt/riscv/code/
keyan@keyan-u22:/opt/riscv/code/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 "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 ?? ()
(gdb) █
```

- 左侧：让 qemu 监听1234端口，等待 gdb 连接
- 右侧：启动 GDB，加载内核镜像并连接到 qemu

我们可以看到右侧输出了一些内容：

- 首先是：GDB版本与配置，运行在 x86_64 Linux 主机，调试 riscv64-unknown-elf 目标
- 之后是：Reading symbols from bin/kernel... GDB正从 bin/kernel 中读取调试符号，并确认目标结构为 RISC - V 64位
- 最后是：连接到了 qemu 的1234端口，进入远程调试模式；0x0000000000001000 in ?? () 显示当前 PC 指向地址 0x1000，?? 表示无法识别函数名，但此处没有调试符号。

在 QEMU 模拟的这款 RISC - V 处理器中，将复位向量地址初始化为 0x1000，再将 PC 初始化为该复位地址，因此处理器将从此处开始执行复位代码

① CPU从 0x1000 进入 MROM

我们输入命令 x/10i \$pc 可以查看当前程序计数器 \$pc 指向的内存区域，知道正在执行和以后要执行的10条指令。我们写一些注释：

```
1 (gdb) x/10i $pc
2 => 0x1000: auipc    t0,0x0 #将pc高20位与0x0拼接存入t0,即t0=pc= 0x1000
3      0x1004: addi    a1,t0,32 #a1=t0+32=0x1020
4      0x1008: csrr    a0,mhartid #读取机器模式hart ID,存入寄存器a0=0x0
5      0x100c: ld      t0,24(t0) #从t0+24处加载8字节数据,t0=0x80000000
6      0x1010: jr      t0 #跳转到t0存储的地址
7      ...
8 (gdb)
```

之后，我们进行单步调试，输入 si，并且每输入一次，就查看一次寄存器中的值 info r t0。

```

(gdb) info r t0
t0                0x0          0
(gdb) si
0x0000000000001004 in ?? ()
(gdb) info r t0
t0                0x1000       4096
(gdb) info r a1
a1                0x0          0
(gdb) si
0x0000000000001008 in ?? ()
(gdb) info r a1
a1                0x1020       4128
(gdb) si
0x000000000000100c in ?? ()
(gdb) info r a0
a0                0x0          0
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0                0x80000000    2147483648
(gdb) si
0x0000000008000000 in ?? ()
(gdb)

```

② 跳转到 0x80000000

我们再输入 `x/10i 0x80000000`，对这个地址处的代码进行反汇编，显示10条指令。该地址处加载的是作为 boot loader 的 `OpenSBI.bin`，如下，我们写一写注释来理解：

```

1  (gdb) x/10i 0x80000000
2  => 0x80000000: csrr    a6,mhartid  #读取机器模式hart ID,并存入a6
3      0x80000004: bgtz    a6,0x80000108 #若a6大于0,跳转,适用于多核场景
4      0x80000008: auipc   t0,0x0    #t0=pc,获取当前地址
5      0x8000000c: addi    t0,t0,1032  #t0=t0+1032
6      0x80000010: auipc   t1,0x0    #t1=pc
7      0x80000014: addi    t1,t1,-16  # t1=t1-16
8      0x80000018: sd      t1,0(t0)  #将t1的值存储到t0+0(即t0)指向的地址
9      0x8000001c: auipc   t0,0x0    #将PC存入t0
10     0x80000020: addi    t0,t0,1020 #t0=t0+1020
11     0x80000024: ld      t0,0(t0)  #从内存地址t0+0处加载8字节数据,存入t0
12  (gdb)

```

- 这些指令是RISC-V 平台的底层固件（OpenSBI）的初始化逻辑，判断是否是多核场景，计算并操作内存地址，用于加载或存储数据，为内核启动做准备

③ OpenSBI初始化并加载内核到 0x80200000,进入 kern_entry

我们执行指令 `break kern_entry`，在此处打下断点

```
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb)
```

接着执行指令 `x/10i 0x80200000`，查看此处的汇编指令，这是内核入口相关的代码

```
1 (gdb) x/10i 0x80200000
2 0x80200000 <kern_entry>: auipc    sp,0x3
3 0x80200004 <kern_entry+4>: mv      sp,sp
4 0x80200008 <kern_entry+8>: j      0x8020000a <kern_init>
5 0x8020000a <kern_init>: auipc    a0,0x3
6 0x8020000e <kern_init+4>: addi    a0,a0,-2
7 0x80200012 <kern_init+8>: auipc    a2,0x3
8 0x80200016 <kern_init+12>: addi    a2,a2,-10
9 0x8020001a <kern_init+16>: addi    sp,sp,-16
10 0x8020001c <kern_init+18>: li      a1,0
11 0x8020001e <kern_init+20>: sub     a2,a2,a0
12 (gdb) c
13 Continuing.
```

- 其中 `0x80200000 <kern_entry>: auipc sp,0x3` 就是我们的练习一中 `la sp, bootstacktop` 实际执行的机器指令，将当前PC的高20位与 `0x3` 拼接，结果存入栈指针寄存器 `sp`，即初始化内核栈的栈顶。
- 其中 `0x80200008 <kern_entry+8>: j 0x8020000a <kern_init>` 对应的是 `tail kern_init`，无条件跳转，且不修改 `ra` 即返回寄存器的值。

我们通过 `si` 单步执行和 `disassemble` 也可以验证出来：

下图是在 `la sp, bootstacktop` 反汇编：

```
(gdb) disassemble
Dump of assembler code for function kern_entry:
=> 0x0000000080200000 <+0>: auipc    sp,0x3
    0x0000000080200004 <+4>: mv      sp,sp
    0x0000000080200008 <+8>: j      0x8020000a <kern_init>
End of assembler dump.
(gdb)
```

下图是在 `tail kern_init` 处反汇编：

```

(gdb) si
kern_init () at kern/init/init.c:9
9      memset(edata, 0, end - edata);
(gdb) disassemble
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>:      auipc    a0,0x3
0x000000008020000e <+4>:      addi     a0,a0,-2 # 0x80203008
0x0000000080200012 <+8>:      auipc    a2,0x3
0x0000000080200016 <+12>:     addi     a2,a2,-10 # 0x80203008
0x000000008020001a <+16>:     addi     sp,sp,-16
0x000000008020001c <+18>:     li       a1,0
0x000000008020001e <+20>:     sub     a2,a2,a0
0x0000000080200020 <+22>:     sd      ra,8(sp)
0x0000000080200022 <+24>:     jal     ra,0x802004b6 <memset>
0x0000000080200026 <+28>:     auipc    a1,0x0
0x000000008020002a <+32>:     addi     a1,a1,1186 # 0x802004c8
0x000000008020002e <+36>:     auipc    a0,0x0
0x0000000080200032 <+40>:     addi     a0,a0,1210 # 0x802004e8
0x0000000080200036 <+44>:     jal     ra,0x80200056 <cprintf>
0x000000008020003a <+48>:     j       0x8020003a <kern_init+48>
End of assembler dump.
(gdb)

```

- 我们可以看到调试器在此处，`memset(edata, 0, end - edata)` 是将已初始化数据段的结束地址 (`edata`) 到未初始化数据段的结束地址 (`end`) 之间清零，即初始化 BSS 段。

接着我们输入 `c`，执行到断点处，可以看到左侧终端输出，说明这时候 OpenSBI 已经启动

```

keyan@keyan-u22:/opt/riscv/code/labcode/lab1$ make debug
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)

(gdb) info r a0
a0      0x0      0
(gdb) si
0x00000000000001010 in ?? ()
(gdb) info r t0
t0      0x80000000      2147483648
(gdb) x/10i 0x80000000
=> 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)
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) x/10i 0x80200000
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
(gdb) c
Continuing.

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

```

补充：

我们也可以使用 `x/10x $sp` 来查看 `sp` 指向的内存区域即栈顶地址 `bootstacktop`，可以看到，在未初始化之前，是杂乱的数据，肯是上一次被使用后的数据，或者是硬件上电后的随机值。

```
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) x/10x $sp
0x8001bd80:      0x8001be00      0x00000000      0x8001be00      0x00000000
0x8001bd90:      0x46444341      0x55534d49      0x00000000      0x00000000
0x8001bda0:      0x00000000      0x00000000
(gdb)
```

"keyan-u22" 14:02 09-10月 -25

我们单步执行，发现执行过 `la sp, bootstacktop` 之后，对栈顶附近的内存进行了初始化清零，位后续函数调用提高干净的栈空间，用实验验证出来我们的练习。

```
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) x/10x $sp
0x80203000 <SBI_CONSOLE_PUTCHAR>:      0x00000001      0x00000000      0x00
000000 0x00000000
0x80203010:      0x00000000      0x00000000      0x00000000      0x00000000
0x80203020:      0x00000000      0x00000000
(gdb)
```

④ 调用 kern_init(), 并输出信息

```
keyan@keyan-u22:/opt/riscv/code/labcode/lab1$ make debug
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: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

```
7          la sp, bootstacktop
(gdb) disassemble
Dump of assembler code for function kern_entry:
=> 0x0000000080200000 <+0>: auipc sp,0x3
0x0000000080200004 <+4>: mv sp,sp
0x0000000080200008 <+8>: j 0x8020000a <kern_init>
End of assembler dump.
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7          la sp, bootstacktop
(gdb) si
9          tail kern_init
(gdb) si
kern_init () at kern/init/init.c:9
9          memset(edata, 0, end - edata);
(gdb) disassemble
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>: auipc a0,0x3
0x000000008020000e <+4>: addi a0,a0,-2 # 0x80203008
0x0000000080200012 <+8>: auipc a2,0x3
0x0000000080200016 <+12>: addi a2,a2,-10 # 0x80203008
0x000000008020001a <+16>: addi sp,sp,-16
0x000000008020001c <+18>: li a1,0
0x000000008020001e <+20>: sub a2,a2,a0
0x0000000080200020 <+22>: sd ra,8(sp)
0x0000000080200022 <+24>: jal ra,0x802004b6 <memset>
0x0000000080200026 <+28>: auipc a1,0x0
0x000000008020002a <+32>: addi a1,a1,1186 # 0x802004c8
0x000000008020002e <+36>: auipc a0,0x0
0x0000000080200032 <+40>: addi a0,a0,1210 # 0x802004e8
0x0000000080200036 <+44>: jal ra,0x80200056 <cprintf>
0x000000008020003a <+48>: j 0x8020003a <kern_init+48>
End of assembler dump.
(gdb) break kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 9.
(gdb) continue
Continuing.
```

"keyan-u22" 13:54 09-10月 -25

在 `kern_init` 的汇编代码的最后，调用 `cprintf` 进行打印，跳转到自身地址，进入死循环，内核初始化完成！

三、实验&原理对应点

1. 复位向量与 MROM

实验：PC= 0x1000 起步，执行少量固件代码。

原理：Reset Vector 与 Boot ROM 提供可信起点。

关系/差异：实验用固定地址验证起跳。原理解释“可信引导链”。

2. OpenSBI 作为固件层

实验：从 0x80000000 进入 OpenSBI，做早期初始化。

原理：M 模式固件向 S 模式内核提供 SBI 接口与抽象。

关系/差异：实验看到跳转与少量汇编。原理关注分层与职责边界。

3. 内核装载地址与内存布局

实验：内核被放到 0x80200000，入口为 kern_entry。

原理：链接脚本决定段布局与对齐，物理地址与虚拟地址映射策略。

关系/差异：实验验证“放哪儿”。原理解释“为何这样放”。

4. 栈初始化 `la sp, bootstacktop`

实验：显式设定内核栈顶，保障后续 C 代码可用。

原理：ABI 调用约定依赖有效栈帧与对齐。

关系/差异：实验是一次性设定。原理涵盖中断栈与异常栈等多栈情形。

5. 尾调用 `tail kern_init`

实验：无返回跳转到 C 入口，释放当前帧。

原理：尾调用优化与调用约定，控制流显式移交。

关系/差异：实验直观可见跳转。原理说明为何可省返回开销。

6. BSS 清零与运行时环境

实验：memset(edata,0,end-edata) 完成 BSS 初始化。

原理：C 运行时要求未初始化全局变量为零。

关系/差异：实验看到具体指令与效果。原理规定语言语义与装载责任。

7. QEMU+GDB 远程调试

实验：make debug/make gdb，断点、单步、反汇编与寄存器观察。

原理：DWARF 符号与断点机制，指令级可观测性与可重复性。

关系/差异：实验掌握操作。原理解释调试器与被调试目标的协议。

8. 入口汇编与 C 运行时过渡

实验：从 entry.S 准备环境再跳入 kern_init。

原理：从极简运行时过渡到抽象更高的内核子系统初始化。

关系/差异：实验聚焦“最小可跑”。原理展开内存、时钟、中断等初始化序列。

四、本实验未覆盖的OS重要原理

1. 进程与线程抽象，调度算法与上下文切换，时钟中断驱动调度。
2. 虚拟内存与页表管理，多级页表，TLB，缺页异常与内存保护。
3. 同步与并发控制，自旋锁，互斥量，屏障，死锁分析。
4. 系统调用接口设计，用户态到内核态的陷入与返回路径。
5. 中断子系统与驱动模型，PLIC/CLINT 细节与中断优先级。
6. 文件系统与存储栈，VFS，日志文件系统，页缓存与回写策略。

7. 安全与隔离，最小特权，能力机制，ASLR 与 W^X 等内存策略。
8. I/O 子系统与 DMA，零拷贝路径与缓存一致性。
9. NUMA 与缓存一致性协议在多核上的性能影响。
10. 用户态装载器与动态链接，ELF 解析与重定位。