

# 南开大学操作系统Lab0.5实验报告

学号：2110598 姓名：许宸  
学号：2113384 姓名：刘新宇  
学号：2112487 姓名：刘轩宇

## 南开大学操作系统Lab0.5实验报告

实验题目 — Lab 0.5 最小可执行内核

实验目的

环境

软件环境

实验步骤与内容

- (一) 初始化
- (二) 运行 0x1000 指令
- (三) 运行 0x1004 指令
- (四) 运行 0x1008 指令
- (四) 运行 0x1010 指令
- (五) 运行jr跳转
- (六) 执行断点

CHATGPT:

第一部分

第二部分

第三部分

- (七) 运行

结论分析与体会

## 实验题目 — Lab 0.5 最小可执行内核

### 实验目的

实验0.5主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。

本章将学到：

- 使用 链接脚本 描述内存布局
- 进行 交叉编译 生成可执行文件，进而生成内核镜像
- 使用 OpenSBI 作为 bootloader 加载内核镜像，并使用 Qemu 进行模拟
- 使用 OpenSBI 提供的服务，在屏幕上格式化打印字符串用于以后调试

### 环境

#### 软件环境

ubuntu22.04

# 实验步骤与内容

根据资料，最小可执行内核的执行流为：

加电 -> OpenSBI启动 -> 跳转到 0x80200000 (kern/init/entry.S) -> 进入 kern\_init() 函数 (kern/init/init.c) -> 调用 cprintf() 输出一行信息 -> 结束。

## (一) 初始化

执行gdb调试后，运行下列代码查看指令计数器后的30条指令：

```
(gdb) x/15i $pc
```

结果如下：

```
=> 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: 0x8000
    0x101c: unimp
    0x101e: unimp4
    0x1020: addi     a2,sp,724
    0x1022: sd      t6,216(sp)
    0x1024: unimp
    0x1026: addiw    a2,a2,3
```

下面我们开始分析 0x1000-0x1010 处的代码：

```
0x1000: auipc    t0,0x0
```

**解释：**auipc 是 "Add Upper Immediate to PC" 的缩写，它将立即数（这里是0x0）左移20位，然后加到程序计数器（PC）的值上，结果存储在 t0 寄存器中。在这里，它只是将当前PC的值（0x1000）存储到 t0 寄存器中，因为立即数是0。

```
0x1004: addi     a1,t0,32
```

**解释：**addi 是 "Add Immediate" 的缩写，它将一个立即数（这里是32）加到一个寄存器的值上（这里是 t0），结果存储在另一个寄存器中（这里是 a1）。在这里，它将 t0 寄存器的值加32，然后存储结果到 a1 寄存器中。

```
0x1008: csrr     a0,mhartid
```

**解释：**csrr 是 "Control and Status Register Read" 的缩写，它读取一个CSR（Control and Status Register）的值并将其存储在一个寄存器中。在这里，它读取 mhartid 寄存器的值（它包含当前硬件线程的ID）并将其存储在 a0 寄存器中。

```
0x100c: ld    t0,24(t0)
```

**解释：**ld 是 "Load Doubleword" 的缩写，它从内存中加载一个双字（64位）到一个寄存器中。在这里，它从 t0 寄存器指向的地址加24的位置加载一个双字，并将其存储在 t0 寄存器中。

```
0x1010: jr    t0
```

**解释：**jr 是 "Jump Register" 的缩写，它将程序计数器（PC）设置为一个寄存器的值。在这里，它将 PC 设置为 t0 寄存器的值，从而导致程序跳转到 t0 寄存器指向的地址。

## (二) 运行 0x1000 指令

下面，我们将运行第一条指令，在运行指令前，首先运行下面的指令检查各寄存器的值：

```
info register
```

结果如下（部分）：

t0	0x0000000000000000	0
a0	0x0000000000000000	0
a1	0x0000000000000000	0
pc	0x0000000000001000	4096

在运行前t0的值为0，指令寄存器pc为 0x0000000000001000，根据汇编代码，预期t0的值将变为 0x0000000000001000

运行 si 执行第一条指令，接着用 info 指令查看寄存器的状态：

```
(gdb) info r t0
t0          0x0000000000001000    4096
(gdb) info r a0
a0          0x0000000000000000    0
(gdb) info r a1
a1          0x0000000000000000    0
(gdb) info r pc
pc          0x0000000000001004    4100
```

实验结果符合预期，进行下一步实验。

## (三) 运行 0x1004 指令

运行指令并检查寄存器的值：

```
(gdb) info r t0
t0          0x0000000000001000    4096
(gdb) info r a0
a0          0x0000000000000000    0
(gdb) info r a1
a1          0x0000000000001020    4128
(gdb) info r pc
pc          0x0000000000001008    4104
```

`a1=0x1000+32(iimm)=0x1020`，进行下一步实验

## (四) 运行 0x1008 指令

运行指令后，发现 `a0` 的值没变，检查 `mhartid` 寄存器的值（它包含当前硬件线程的ID）：发现是0

```
(gdb) info r mhartid
mhartid      0x0000000000000000    0
```

分析原因：

**CHATGPT：**如果硬件线程的ID是0，这意味着这段代码是在第一个硬件线程（或核心）上执行的。在多线程处理器系统中，每个核心或线程都有一个唯一的ID，用于区分它们。在这段代码中，`mhartid` 寄存器被用来存储当前硬件线程的ID。

在这种情况下，第三行代码：

```
0x1008: csrr a0,mhartid
```

将会把0加载到 `a0` 寄存器中，因为 `mhartid` 寄存器的值是0。

这个信息可能会被用来执行一些特定于线程0的任务或初始化操作。例如，它可以用来确保只有线程0执行某些特定的初始化任务，而其他线程则执行不同的任务或等待线程0完成初始化。

## (四) 运行 0x1010 指令

运行后检查寄存器的值

```
(gdb) info r t0
t0      0x0000000080000000    2147483648
(gdb) info r a1
a1      0x0000000000001020    4128
(gdb) info r pc
pc      0x0000000000001010    4112
```

注意到 `t0` 取的是 `0x1018` 处的值：

```
0x1018: unimp
```

**CHATGPT：**`unimp` 是 RISC-V 汇编语言中的一个指令，它表示 "unimplemented"。这个指令是一个占位符，用于标记尚未实现的指令位置。当处理器遇到这个指令时，它通常会触发一个异常，因为它表示一个未定义的或未实现的操作。

在实际的程序开发过程中，`unimp` 指令可以用作一个临时占位符，允许开发人员在未完成的程序中保留空间，以便稍后添加实际的指令。它也可以用于标记那些应该被更高级别的编译器或汇编器替换的位置。

运行下面的指令

```
(gdb) x/g 0x1018
0x1018: 0x0000000080000000
```

发现 `0x1018` 处的值是 `0x0000000080000000`，证明 `0x1018` 处存在数，实验继续。

## (五) 运行jr跳转

检查寄存器：

```
(gdb) info r t0
t0          0x0000000080000000    2147483648
(gdb) info r a0
a0          0x0000000000000000    0
(gdb) info r a1
a1          0x0000000000001020    4128
(gdb) info r pc
pc          0x0000000080000000    2147483648
```

已跳转至 0x0000000080000000，继续试验。

上述五句代码首先将程序计数器的当前值存储到 `t0` 寄存器中，然后在该值上加32并存储到 `a1` 寄存器中。然后，它读取当前硬件线程的ID并存储到 `a0` 寄存器中。接着，它从内存中加载一个双字到 `t0` 寄存器中，然后跳转到 `t0` 寄存器指向的地址。

思考：为什么要读取当前硬件线程的id？

**CHATGPT：**在多核系统中，通常只有一个核心（通常是核心0）负责执行初始化代码，而其他核心则等待初始化完成。通过读取硬件线程ID，程序可以确定它是在哪个核心上运行，从而决定执行初始化代码还是进入等待状态。

## (六) 执行断点

我们知道第一个应用的位置在 0x80200000 处，我们在此处设置一个断点：

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

根据路径，我们定位到源文件：

```
#include <mmu.h>
#include <memlayout.h>

.section .text,"ax",%progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

这是一个c++内嵌入汇编的文件，可以看出第7行是kern\_entry这一函数的返回。

## CHATGPT:

### 第一部分

```
#include <mmu.h>
#include <memlayout.h>
```

这两行代码是C语言预处理器指令，用于包含两个头文件：`mmu.h` 和 `memlayout.h`。这些文件通常包含有关内存管理单元（MMU）和内存布局的定义和声明。

### 第二部分

```
.section .text,"ax",%progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init
```

这部分是汇编代码，解释如下：

1. `.section .text,"ax",%progbits`：这行指令将代码段定义为“.text”段，该段具有执行和访问权限，并且其内容是程序代码。
2. `.globl kern_entry`：这行指令使 `kern_entry` 标签全局可见，这样其他文件也可以引用它。
3. `kern_entry:`：这是一个标签，标记了代码的入口点。
4. `la sp, bootstacktop`：这行指令将 `bootstacktop` 的地址加载到堆栈指针（sp）寄存器中。`la` 是“加载地址”的简写。

**这一句是在程序的初始化阶段完成的，以设置栈的初始状态，为后续的函数调用和局部变量存储做准备。在实际的系统或内核编程中，这样的操作是非常常见的，因为它帮助建立一个已知和可控的环境，从而避免栈溢出或其他与内存管理相关的错误。**

1. `tail kern_init`：这行指令是一个尾调用到 `kern_init` 函数，它是内核初始化的开始。

### 第三部分

```
.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

这部分也是汇编代码，解释如下：

1. `.section .data`：这行指令将数据段定义为“.data”段，用于存储初始化的全局变量和静态变量。
2. `# .align 2^12`：这是一个注释，表示原来可能使用了 $2^{12}$ 的对齐，但现在被替换为 `PGSHIFT`。
3. `.align PGSHIFT`：这行指令将下一个变量或标签对齐到 `PGSHIFT` 定义的边界。

4. `.global bootstack`: 这行指令使 `bootstack` 标签全局可见。
5. `bootstack::`: 这是一个标签, 标记了 `bootstack` 的开始。
6. `.space KSTACKSIZE`: 这行指令在数据段中分配 `KSTACKSIZE` 字节的空间, 通常用于内核栈。
7. `.global bootstacktop`: 这行指令使 `bootstacktop` 标签全局可见。
8. `bootstacktop::`: 这是一个标签, 标记了 `bootstack` 的顶部。

这段代码主要用于设置内核的初始状态, 包括初始化栈和定义内核入口点。在内核初始化函数 `kern_init` 中, 可能会进行更多的设置和初始化操作。

## (七) 运行

运行 `continue` 后, 在断点处停下, 查看指令:

```
(gdb) x/10i $pc
=> 0x80200004 <kern_entry+4>:  mv    sp,sp
0x80200008 <kern_entry+8>:  j      0x8020000c <kern_init>
0x8020000c <kern_init>:    auipc   a0,0x3
0x80200010 <kern_init+4>:    addi    a0,a0,-4
0x80200014 <kern_init+8>:    auipc   a2,0x3
0x80200018 <kern_init+12>:   addi    a2,a2,-12
0x8020001c <kern_init+16>:   addi    sp,sp,-16
0x8020001e <kern_init+18>:   li      a1,0
0x80200020 <kern_init+20>:   sub     a2,a2,a0
0x80200022 <kern_init+22>:   sd      ra,8(sp)
```

`mv sp,sp` 指令起到占位作用

`0x80200008` 显示将要跳转到 `<kern_init>`, 我们查看 `init.c`

```
#include <stdio.h>
#include <string.h>
#include <sbi.h>
int kern_init(void) __attribute__((noreturn));

int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);
    while (1)
        ;
}
```

`kern_init` 函数执行了打印字符串的操作, 输出 `(THU.CST) os is loading`, 我们将 `THU.CST` 改为 `NKU.CST`:

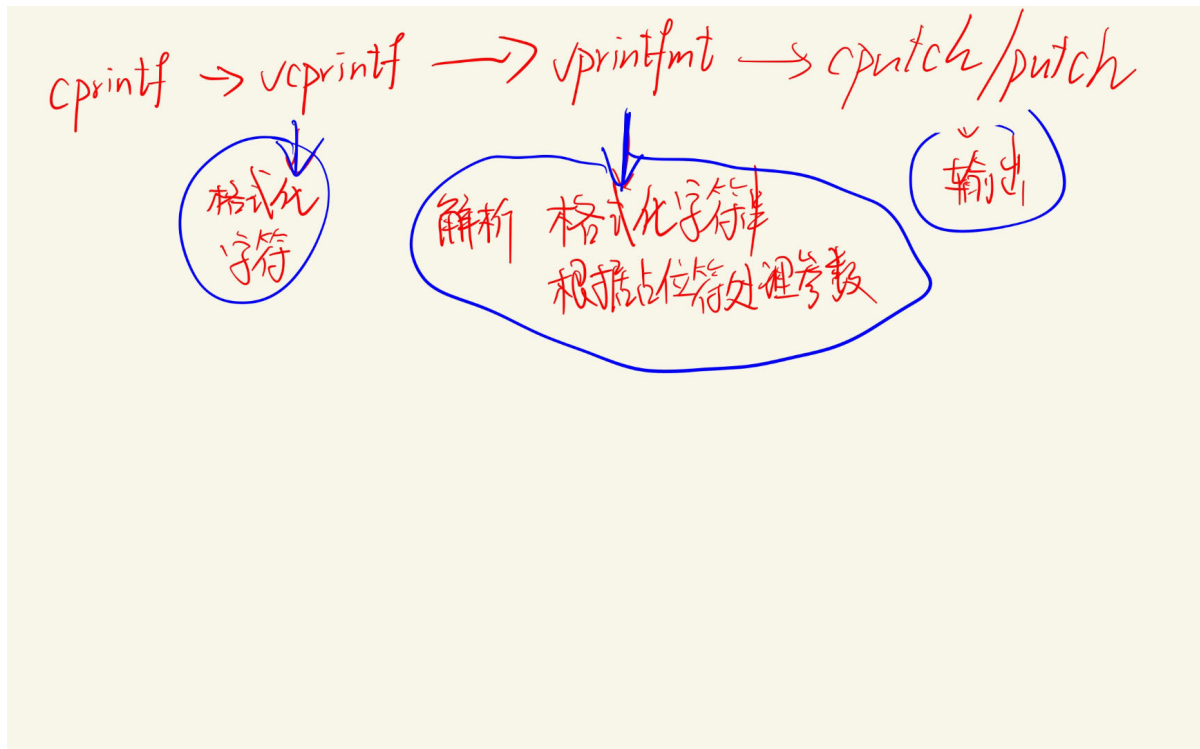
运行下面两条指令:

```
break *0x80200008
break *0x8020000c
```

查看后面的更多代码：

```
0x8020003e <cputch>: addi    sp,sp,-16
0x80200040 <cputch+2>: sd     s0,0(sp)
0x80200042 <cputch+4>: sd     ra,8(sp)
0x80200044 <cputch+6>: mv     s0,a1
0x80200046 <cputch+8>: jal    ra,0x8020008c <cons_putc>
0x8020004a <cputch+12>: lw     a5,0(s0)
0x8020004c <cputch+14>: ld     ra,8(sp)
0x8020004e <cputch+16>: addiw  a5,a5,1
0x80200050 <cputch+18>: sw     a5,0(s0)
0x80200052 <cputch+20>: ld     s0,0(sp)
---Type <return> to continue, or q <return> to quit---
0x80200054 <cputch+22>: addi    sp,sp,16
0x80200056 <cputch+24>: ret
```

后面存在cputch等一系列函数，函数调用结构如下：



OpenSBI输出了 THU.CST，并不是NKU（悲）

原因分析：



```
app@app-virtual-machine: ~/riscv64-ucore-labcodes/riscv64-...
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)
(THU.CST) os is loading ...
```

可能是之前的代码已经加载进内存里了。重新改过一次文件后，启动qemu和gdb重新执行

```
app@app-virtual-machine: ~/riscv64-ucore-labcodes/riscv64-...
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)
(NKU.CST) os is loading ...
```

证明猜想可能是正确的

## 结论分析与体会