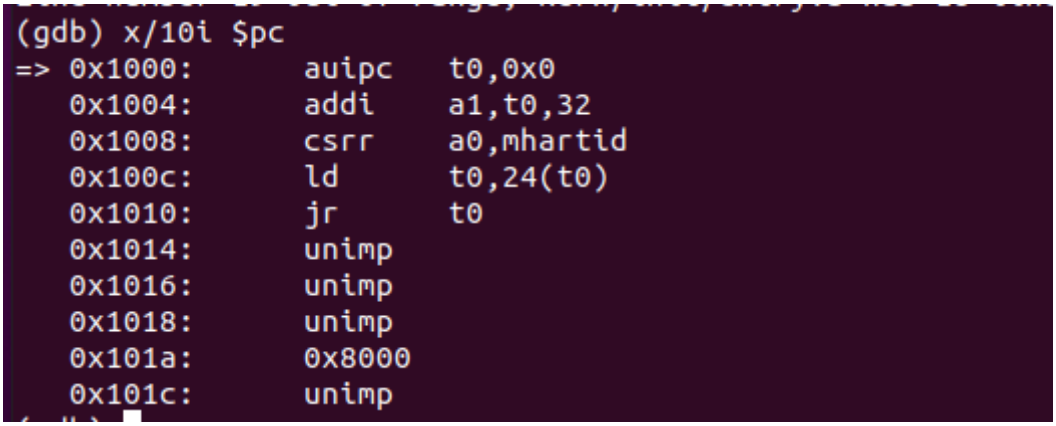
操作系统实验作业——lab0.5

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

加电

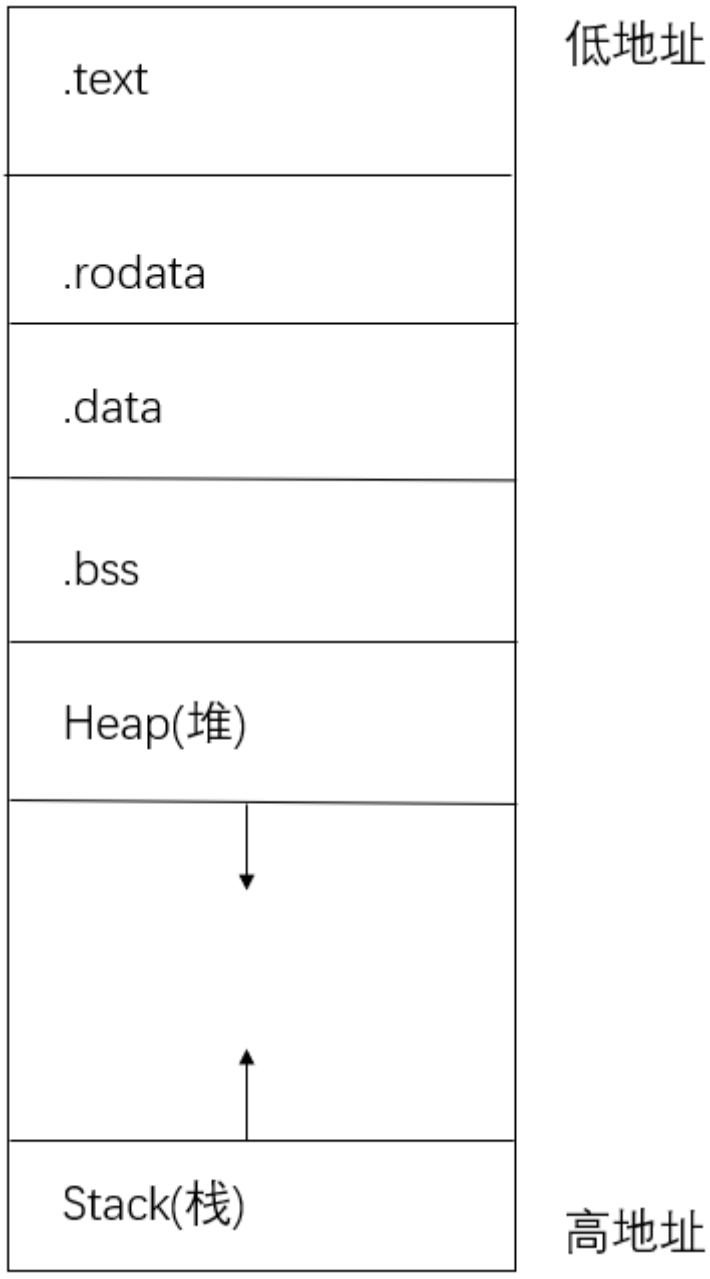
在qemu模拟器中，当启动qemu模拟时，qemu会模拟计算机加电的过程。在qemu执行任何指令之前，将OpenSBI.bin 被加载到物理内存以物理地址 0x80000000 开头的区域上，同时内核镜像 os.bin 被加载到以物理地址 0x80200000 开头的区域上。此时，riscv处理器会将PC置于复位地址处，在本实验中模拟的riscvCPU的复位地址是0x1000。这个过程主要是将计算机系统内的各个部件置于初始状态，然后启动处于物理地址 0x80000000 的Bootloader——OpenSBI.bin。



OpenSBI启动

OpenSBI作为qemu自带的bootloader，OpenSBI启动时会进行一些初始化操作，如初始化RISC-V系统的硬件，包括处理器核心、内存控制器、外部设备等。在可执行文件中，为了正确地和OpenSBI对接，会将内核的指令加载到固定的位置，既0x80200000。所以当初始化工作完成后，会加载OS到内存，既执行0x80200000处的指令，将控制权交给内核。

程序结构



1. **.text 段（代码段）**：包含程序的可执行指令，通常存放汇编代码或编译后的机器代码。
2. **.rodata 段（只读数据段）**：.rodata 段通常包含只读的常量数据，如字符串字面量和其他常量值，因为是只读数据段，所以这些数据在程序运行时不能被修改。
3. **.data段**：.data 段包含被初始化的可读写数据，通常存放程序中的全局变量和静态变量。与.rodata段不同，该段数据在程序运行时可以被读取和修改。
4. **.bss段**：.bss 段包含被初始化为 0 的可读写数据。与 .data 段的不同之处在于，.bss 段只需记录大小和位置，而不需要记录具体的数据，因为它们被初始化为 0。等程序被加载到内存中时，.bss 段才会为他们真正分配空间。
5. **栈（stack）**：栈用于存储程序运行过程中的局部变量、函数调用的上下文信息和执行状态。在内存中，栈通常从高地址向低地址增长，即栈顶在高地址，栈底在低地址。栈的大小在程序运行时会动态变化，栈指针（Stack Pointer）用于指示栈顶的位置。
6. **堆（heap）**：堆用于支持程序运行过程中的动态内存分配，通常用于存储程序在运行时分配的数据，如动态数组、对象等。堆的位置和大小由操作系统管理，程序可以通过堆管理函数（如 malloc 和 free）来进行动态内存分配和释放。堆的地址通常位于 .bss 段之后，但具体位置由操作系统分配。

链接

链接器是编译过程中的一个关键组件，它负责将多个目标文件（通常是 .o 文件）合并成一个最终的可执行文件或共享库。链接器的主要作用包括符号解析、地址分配、重定位和生成最终的输出文件。对于一般来说，输入文件和输出文件都有很多section, 链接脚本(linker script)的作用，就是描述怎样把输入文件的section映射到输出文件的section, 同时规定这些section的内存布局。所以通过链接之后，在执行该程序时，内核才能准确地找到每一条该执行地指令和数据。

在本程序中，链接脚本把程序的入口点定义为kern_entry，同时初始化了一个BASE_ADDRESS，之后便从BASE_ADDRESS表示的地址处，逐步加载每个section，在这里首先说明了先把kern_entry放在.text段开头，然后依次完善各个section。

```
/* tools/kernel.ld */

OUTPUT_ARCH(riscv)
ENTRY(kern_entry)  /* 指定程序的入口点，是一个叫做kern_entry的符号。我们之后会在汇编代码里定义它*/

BASE_ADDRESS = 0x80200000;/*定义了一个变量BASE_ADDRESS并初始化 */

SECTIONS
{
 /* Load the kernel at this address: "." means the current address */
 . = BASE_ADDRESS;/*对 "."进行赋值*/
 .text : {
     *(.text.kern_entry) /*把输入中kern_entry这一段放到输出中text的开头*/
     *(.text .stub .text.* .gnu.linkonce.t.*)
 }
 ......
```

程序执行

现在我们通过连接器，将程序地入口定义为了kern_entry，所以我们程序里需要有一个名称为kern_entry的符号。我们在kern/init/entry.S编写一段汇编代码, 作为整个内核的入口点。

在下面这段代码中，我们首先说明了接下来这一段是.text段中地代码，声明kern_entry为全局符号，使得使得链接器能够访问它。在kern_entry标签内部，将bootstacktop加载到sp寄存器中，这里是将栈指针指向bootstacktop处，即内核栈的顶部。之后调用kern_init函数，但在函数返回时不会创建新的函数调用帧，避免了栈的不必要增长。总的来说，在分配好内核栈后，会跳转到kern_init这个真正的入口点，操作系统内核成功启动。

```
.section .text,"ax",%progbits
 .globl kern_entry # 使得ld能够看到kern_entry这个符号所在的位置，globl和global同义
kern_entry:
 la sp, bootstacktop
 tail kern_init
```

```
(gdb) l
1       #include <mmu.h>
2       #include <memlayout.h>
3
4           .section .text,"ax",%progbits
5           .globl kern_entry
6       kern_entry:
7           la sp, bootstacktop
8
9           tail kern_init
10
```

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

0          const char *message = "(THU.CST) os is loading ...\n";
1          cprintf("%s\n\n", message);
2         while (1)
3             ;
4      }
```

函数调用

在我们编写内核时，并不能直接去调用c语言等地库文件，而是要通过ecall指令(environment call)调用OpenSBI为我们提供的接口。有时OpenSBI调用需要像函数调用一样传递参数，这里传递参数的方式也和函数调用一样。从下面这段代码中定义了sbi_call函数，因为c语言不能直接调用ecall，需要通过使用内联汇编来首先调用。

我们可以发现，x17寄存器存储的sbi_type，即调用对应SBI编号的函数，在x10、x11、x12三个寄存器中依次储存三个参数。使用 ecall 指令触发 SBI 调用，将控制权交给 OpenSBI 执行。之后函数的返回值储存在x10寄存器中。函数调用时，通常会将返回地址存储在ra寄存器中。这个寄存器会被自动保存和恢复，以保证函数返回后能够正确回到调用者。与x86中函数调用不同，riscv参数是存储在寄存器中。在函数返回时，返回值通常会存储在a0寄存器中，同时将控制流返回到存储在ra寄存器中的返回地址，从而继续执行调用者的代码。

```
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
arg2) {
 uint64_t ret_val;
 __asm__ volatile (
     "mv x17, %[sbi_type]\n"
     "mv x10, %[arg0]\n"
     "mv x11, %[arg1]\n"
     "mv x12, %[arg2]\n"   //mv操作把参数的数值放到寄存器里
     "ecall\n"    //参数放好之后，通过ecall，交给OpenSBI来执行
     "mv %[ret_val], x10"
     //OpenSBI按照riscv的calling convention,把返回值放到x10寄存器里
     //我们还需要自己通过内联汇编把返回值拿到我们的变量里
     : [ret_val] "=r" (ret_val)
     : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
[arg2] "r" (arg2)
     : "memory"
 );
 return ret_val;
}
```

函数封装

我们一层一层地封装sbi_console_putchar函数，使得最后得到cprintf()函数地功能和C标准库的printf()基本相同。

```
// kern/driver/console.c
#include <sbi.h>
#include <console.h>

void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }

// kern/libs/stdio.c
#include <console.h>
#include <defs.h>
#include <stdio.h>

/* HIGH level console I/O */

/* *
 * cputch - writes a single character @c to stdout, and it will
 * increace the value of counter pointed by @cnt.
 * */
static void cputch(int c, int *cnt) {
    cons_putc(c);
    (*cnt)++;
}
/* cputchar - writes a single character to stdout */
void cputchar(int c) { cons_putc(c); }

int cputs(const char *str) {
    int cnt = 0;
    char c;
    while ((c = *str++) != '\0') {
        cputch(c, &cnt);
    }
    cputch('\n', &cnt);
    return cnt;
}
```

```
35       * written to stdout.
36       * */
37      int cprintf(const char *fmt, ...) {
38          va_list ap;
39          int cnt;
40          va_start(ap, fmt);
41          cnt = vcprintf(fmt, ap);
42          va_end(ap);
43          return cnt;
44      }
```

```
46      /* cputchar - writes a single character to stdout */
47      void cputchar(int c) { cons_putc(c); }
48
49      /* *
50       * cputs- writes the string pointed by @str to stdout and
51       * appends a newline character.
52       * */
53      int cputs(const char *str) {
54          int cnt = 0;
(gdb)
55          char c;
56          while ((c = *str++) != '\0') {
57              cputch(c, &cnt);
58          }
59          cputch('\n', &cnt);
60          return cnt;
61      }
```

```
63      /* getchar - reads a single non-zero character from stdin */
64      int getchar(void) {
(gdb)
65          int c;
66          while ((c = cons_getc()) == 0) /* do nothing */;
67          return c;
68      }
```

```
/* cons_putc - print a single character @c to console devices */
void cons_putc(int c) { sbi_console_putchar((unsigned char)c); }
```

```
32      void sbi_console_putchar(unsigned char ch) {
33          sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
34      }
```

思考

地址无关代码

地址无关代码（Position-Independent Code，PIC）是一种编写方式，是指可在主存储器中任意位置正确地运行，而不受其绝对地址影响的一种机器码。PIC广泛使用于共享库，使得同一个库中的代码能够被加载到不同进程的地址空间中。PIC还用于缺少内存管理单元的计算机系统中，使得操作系统能够在单一的地址空间中将不同的运行程序隔离开来。使用gcc编译位置无关代码的话，需要加上-fPIC编译选项。

地址无关代码能够在不做修改的情况下被复制到内存中的任意位置。这一点不同于重定位代码，因为重定位代码需要经过链接器或加载器的特殊处理才能确定合适的运行时内存地址。 地址无关代码需要在源代码级别遵循一套特定的语义，并且需要编译器的支持。那些引用了绝对内存地址的指令（比如绝对跳转指令）必须被替换为PC相对寻址指令。这些间接处理过程可能导致PIC的运行效率下降，但是目前大多数处理器对PIC都有很好的支持，使得这效率上的这一点点下降基本可以忽略。

如果程序使用静态库时，当编译该程序时，对于会对所有的静态库符号进行重定位，这样当程序加载到内存时，才能进行调用。而如果程序链接动态库的话，因为动态库并没有和程序联编，这就导致了链接器不能为动态库的符号经行重定位。只有动态库被系统加载的时候，各个符号的地址才被确定，而动态库的符号重定位便由动态链接器载入器完成。

1. GOT

GOT（Global Offset Table）表中每一项都是本运行模块要引用的一个全局变量或函数的地址。可以用GOT表来间接引用全局变量、函数，也可以把GOT表的首地址作为一个基准，用相对于该基准的偏移量来引用静态变量、静态函数。由于加载器不会把运行模块加载到固定地址，在不同进程的地址空间中，各运行模块的绝对地址、相对位 置都不同。这种不同反映到GOT表上，就是每个进程的每个运行模块都

有独立的GOT表，所以进程间不能共享GOT表。

2. PLT

PLT（Procedure Linkage Table）表每一项都是一小段代码，对应于本运行模块要引用的一个全局函数。第一次调用以后，GOT表项已指向函数的正确入口。以后再有对该函数的调用，跳到PLT表后，不再进入加载器，直接跳进函数正确入口了。从性能上分析，只有第一次调用才要加载器作一些额外处理，这是完全可以容忍的。还可以看出，加载时不用对相对跳转的代码进行修补，所以整个代码段都能在进程间共享。