

# 技术文档

## 虚拟机环境搭建

因为基于RISCV体系结构，在x86平台下，需要虚拟机和交叉编译器。

### 虚拟机

使用WSL (Windows Subsystem for Linux) ,使用系统镜像为 Ubuntu-22.04

### 工具链

因为RISCV官方仓库的交叉编译工具链很大，编译需要很久，且很多东西并不需要，所以使用了Ubuntu仓库提供的工具链，即 riscv64-linux-gnu-toolchain

## RISCV体系结构 和qemu 跳转

RISC-V 架构有三种主要的操作模式：

**用户模式** (User mode) 用户模式是最基本的执行模式，应用程序运行在此模式下，无法访问特权指令或系统资源

**超级用户模式** (Supervisor mode) 超级用户模式通常由操作系统内核使用，允许访问更多的硬件资源和特权指令，用于管理系统资源；

**机器模式** (Machine mode) 机器模式是最高权限模式，通常由固件或引导程序使用，具有完全访问硬件的权限，用于初始化系统并管理硬件的基本操作。

在 QEMU 中，默认使用 OpenSBI 启动 RISC-V 系统时，通常是通过加载一个预编译的固件 (OpenSBI) 来初始化机器和启动内核。OpenSBI (Open RISC-V Supervisor Binary Interface) 是一个 RISC-V 平台初始化程序，它为 RISC-V 运行时提供了基本的硬件抽象层 (HAL) 和系统调用接口。启动过程中，QEMU 会加载 OpenSBI 固件，然后通过它加载并启动 Linux 内核。常见的启动方式包括使用 QEMU 的 `-bios` 选项指定 OpenSBI 固件文件。

可以使用OpenSBI进行一些机器模式(M-mode)的操作，但我找了资料参考封装库，最后只能使用一个 `sbi_console_get_char()` 的函数，其他的测试下来均不能用，**只能放弃，选择不使用SBI。**

因此这里qemu的命令为：

```
qemu-system-riscv64 -machine virt -m 128M -bios none -kernel kernel.elf -nographic
```

**kernel.elf**为代码编译链接后的可执行文件，在链接脚本中只将起始地址设为0x80000000。

```
//linker.ld
SECTIONS {
    . = 0x80000000;
    .text : { *(.text) }
    .bss : { *(.bss) }
    . = ALIGN(4K);
    _stack_top = .;
}
```

0x80000000 是qemu启动后自动跳转到的地址，自己编写的内核代码应该从这里开始执行

## UART(Universal Asynchronous Receiver/Transmitter，通用异步接收/发送器)

UART是一种常见的串行通信协议，用于在计算机和外部设备（如调试终端、外部硬件等）之间传输数据。

qemu 在0x10000000处映射了UART设备

可以通过 UART 输出启动信息

由于还没有封装UART，可以先简单的将字符串传输到地址0x10000000,如下：

```
const char *str = "Enter start....\n";
while (*str) {
    *(volatile char *)0x10000000 = *str++; // 使用 UART 打印
}
```

已经具备简单的信息输出，可以开始构建最小的机器启动了

## 1.最小机器启动

```
// entry.S

.section .text
.global _start
_start:
    la sp, stack_top
    call init

.section .bss
.global stack
.align 4
```

```
stack:
    .skip 4096 # 4KB 栈空间
stack_top:
```

在entry.S中，只做最简单，定义一个\_start函数，为分配4KB的栈空间，然后跳转到init函数中（在start.c中），此时我们就进入了start.c处于M-mode，可以执行riscv.h中定义的函数，为机器进行初始化。

由于我们只是最开始的启动，所有不做过多的内容，start.c中，就仅仅打印一些信息，然后跳转到main.c

```
// start.c

extern void main();
void init(){
    // 使用 UART 打印信息
    const char *str = "Enter start....\n";
    while (*str) {
        *(volatile char *)0x10000000 = *str++;
    }
    // 关闭分页
    w_satp(0);

    int id = r_mhartid();
    w_tp(id);

    // 欺骗cpu上一个状态为S-mode
    uint64 x = r_mstatus();
    x &= ~MSTATUS_MPP_MASK;
    x |= MSTATUS_MPP_S;
    w_mstatus(x);

    // 返回到main函数
    w_mepc((uint64)main);
    asm volatile("mret");
}
```

同样在main.c中也打印一下信息

```
riscv64-linux-gnu-as kernel/trap.S -o kernel/trap.o
riscv64-linux-gnu-ld -o kernel.elf kernel/entry.o kernel/cpu.o kernel/main.o kernel/plic.o kernel/
el/trap.o -T linker.ld
Enter start....
Enter main, cpuid: 0
```

可以看到，最小的机器已经启动起来了，

## 2.UART封装和printf函数实现

由于什么直接使用UART进行打印的方式不是很灵活，所有可以将UART封装，再封装出一个简单的printf()函数，供我们调试使用。

### [UART参考标准](#) | [UART规范](#)

参考上述的规范文件和xv6的UART实现，对UART进行简单的封装和初始化（UART.c文件）

### 实现自旋锁

因为UART并行输出的问题，所以实现了简单的自旋锁进行控制。

**自旋锁，即当锁被他人持有时，自己会一直等待，直到可以获取到锁，再使用**

```
// lock.h

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

void spinlock_init(spinlock_t* lk, char* name);
void spinlock_acquire(spinlock_t* lk);
void spinlock_release(spinlock_t* lk);
bool spinlock_holding(spinlock_t* lk);
```

### 实现printf()函数

简单的printf(),考虑到后面的使用，暂时仅仅支持

- 1. %d 整型的格式
- 1. %s 字符串
- 1. %p 指针或者地址，为后面内存分配做铺垫
- 1. %x 整型的十六进制输出（不重要）

### 如何获取到格式化的可变参数？

在C语言的 <stdarg.h>中定义了用于处理格式化参数的宏

1. `va_start`：在访问可变参数前调用该宏，它初始化一个 `va_list` 类型的变量，指向参数列表的第一个参数。

2. `**`va_arg`**`: 用于访问可变参数，每次调用时返回下一个参数，并将指针指向下一个参数
2. `**`va_end`**`: 在访问完所有可变参数后调用，用于清理 ``va_list`` 变量

使用上述宏定义，可以简单的完成可变参数的获取和使用。

则**printf()**可以简单实现为：

```
void printf(char *fmt, ...)
{
    //获取锁
    spinlock_acquire(&print_lk);
    //可变参数初始化
    const char *ptr = fmt;
    va_list args;
    va_start(args, fmt);

    while (*ptr != '\0')
    {
        //格式化
        if(*ptr == '%'){
            ptr++;
            if (*ptr == 'd')           // int
            {

            }

            if (*ptr == 'x') //hex
            {

            }

            if (*ptr == 'p') // ptr
            {

            }

            if (*ptr == 's') //str
            {

            }

        }else{
            //非格式化字符
            uart_putchar_sync(*ptr);
        }
        ptr++;
    }
    //清理可变参数和释放锁
    va_end(args);
    spinlock_release(&print_lk);
}
```

实现相应的类型转字符串函数，在对应的格式化里面填入输出逻辑即可

## 附加：

1. **assert(bool exp, char\* str)**: 用于判断条件 **exp** 是否成立，不成立则输出错误信息并终止
2. **panic(char\* str, char\* file, int line)** : 输出错误信息和发生错误的文件及行号并终止  
对panic再加一层宏定义

```
#define PANIC(msg) panic(msg, __FILE__, __LINE__)
```

即 利用编译器提供的**FILE** 和**LINE**两个宏定义，帮助我们确定错误信息和发生错误的地点

## 3. 内存管理

采用SV39的页表管理模式，页面大小**PAGESIZE**为4KB

### SV39 的工作原理：

**sv39** 使用三级页表结构来管理虚拟地址到物理地址的映射。具体来说，它将虚拟地址划分为几个部分，分别用来索引不同级别的页表。每个虚拟地址的不同部分对应页表的不同层级，具体格式如下：

1. **虚拟地址结构**：虚拟地址（VADDR）在 **sv39** 中是 39 位，可以分为：
  - **VPN[2] (虚拟页号 2)**：虚拟地址的高 9 位，用于索引页目录。
  - **VPN[1] (虚拟页号 1)**：接下来的 9 位，用于索引页表。
  - **VPN[0] (虚拟页号 0)**：接下来的 9 位，用于索引页表项。
  - **页内偏移量**：最低的 12 位表示页内偏移。
  - 使用**uint64**存储，39到63 设置为0
2. **三级页表**：
  - **顶级页表**：最高 9 位索引，用于选择页全局目录。
  - **次级页表**：次高 9 位索引，用于选择上级页表。
  - **低级页表**：中间的 9 位索引，用于选择页表项。
  - **Page Offset**：低 12 位是页面内的偏移量，用于定位页内的具体地址。

### 页表项（PTE）：

每个页表项（PTE）通常包括：

- **有效位**（Valid bit）：指示页表项是否有效。
- **物理页号**（Physical Page Number）：映射的物理地址。
- **权限字段**：指示该页的访问权限（如只读、读写、用户模式或超级用户模式等）。
- 使用**uint64**存储，定义为

PTE定义:

```
reserved + PPN[2] + PPN[1] + PPN[0] + RSW + D A G U X W R V 共64bit
      10      26      9      9      2      1 1 1 1 1 1 1
      // 页面权限控制
#define PTE_V (1 << 0) // valid
#define PTE_R (1 << 1) // read
#define PTE_W (1 << 2) // write
#define PTE_X (1 << 3) // execute
#define PTE_U (1 << 4) // user
#define PTE_G (1 << 5) // global
#define PTE_A (1 << 6) // accessed
#define PTE_D (1 << 7) // dirty
```

## a. 物理内存

对物理内存的管理需要分页和对内存区域进行划分

每页使用页节点(page\_node\_t)维护, 对内存区域使用分配区 (alloc\_zone\_t) 维护

```
//页节点, 使用单链表进行管理
typedef struct page_node
{
    struct page_node* next;
}page_node_t;
//分配区
typedef struct alloc_zone
{
    uint64 start;
    uint64 end;
    spinlock_t mem_lk;
    uint32 alloc_num;
    page_node_t alloc_head;
}alloc_zone_t;

//维护两个分配区, 分别供kernel和user使用
static alloc_zone_t kernel_zone,user_zone;
// 初始化内存区
void memset(void* begin,uint8 value,uint32 n);

//init,alloc,free
void pmem_init();           //初始系统, 只调用一次
void* pmem_alloc(bool inkernel); // 申请物理页
void pmem_free(bool inkernel,uint64 addr); // 释放物理页

//测试输出用, 不重要
void pmem_stat(bool inkernel);
```

## 内存空间划分

```
// memlayout.h
#define KERNEL_DATA 0x80200000

#define ALLOC_START 0x80400000

#define KERNEL_START ALLOC_START
#define KERNEL_PAGES (uint64)0x400
#define KERENL_END KERNEL_START+KERNEL_PAGES*PAGESIZE

#define USER_START KERENL_END

#define USER_PAGES (uint64)0x1000
#define USER_END USER_START+USER_PAGES*PAGESIZE
#define ALLOC_END USER_END

/*
划分
0x80000000 ~ 0x80200000 内核代码区
0x80200000 ~ 0x80400000 内核数据区

KERNEL_START 内核可分配区域开始
分配了 0x400个页面，
即内核分配区域： 0x80400000 ~ 0x80500000

USER_START 用户可分配区域开始
分配了 0x1000个页面，
即用户可分配区域： 0x80500000 ~ 0x80700000

只是先简单的分配，后续可根据需要再更改

*/
```

## 测试

```
printf("-----physical memory test -----\\n");

int* mem[16];
for (int i = 0; i < 16; i++)
{
    mem[i] = pmem_alloc(1);
    memset(mem[i],1,PAGESIZE);
    printf("mem[%d]: %p alloced ,data= %d\\n",i,mem[i],mem[i][0]);
}
```



```

for (int i = 0; i < 16; i++)
{
    pmem_free(1,mem[i]);
    printf("mem[%d]: %p free\n",i,mem[i]);
}

```

```

Enter start...
Enter main, cpuid: 0
-----physical memory test -----
mem[0]: 0x000000080412000 allocated ,data= 16843009
mem[1]: 0x000000080413000 allocated ,data= 16843009
mem[2]: 0x000000080414000 allocated ,data= 16843009
mem[3]: 0x000000080415000 allocated ,data= 16843009
mem[4]: 0x000000080416000 allocated ,data= 16843009
mem[5]: 0x000000080417000 allocated ,data= 16843009
mem[6]: 0x000000080418000 allocated ,data= 16843009
mem[7]: 0x000000080419000 allocated ,data= 16843009
mem[8]: 0x00000008041a000 allocated ,data= 16843009
mem[9]: 0x00000008041b000 allocated ,data= 16843009
mem[10]: 0x00000008041c000 allocated ,data= 16843009
mem[11]: 0x00000008041d000 allocated ,data= 16843009
mem[12]: 0x00000008041e000 allocated ,data= 16843009
mem[13]: 0x00000008041f000 allocated ,data= 16843009
mem[14]: 0x000000080420000 allocated ,data= 16843009
mem[15]: 0x000000080421000 allocated ,data= 16843009
mem[0]: 0x000000080412000 free
mem[1]: 0x000000080413000 free
mem[2]: 0x000000080414000 free
mem[3]: 0x000000080415000 free
mem[4]: 0x000000080416000 free
mem[5]: 0x000000080417000 free
mem[6]: 0x000000080418000 free
mem[7]: 0x000000080419000 free
mem[8]: 0x00000008041a000 free
mem[9]: 0x00000008041b000 free
mem[10]: 0x00000008041c000 free
mem[11]: 0x00000008041d000 free
mem[12]: 0x00000008041e000 free
mem[13]: 0x00000008041f000 free
mem[14]: 0x000000080420000 free
mem[15]: 0x000000080421000 free

```

## 为什么data值是16843009?

因为memset () 内部按字节进行内存设置，而读取时是按int型读取的，即连续的4字节

$$2^0 + 2^8 + 2^{16} + 2^{24} = 16843009$$

## b.虚拟内存

虚拟内存需要完成

1. 页表的实现
2. 虚拟地址到物理地址的转变

```

//vmem.h

//测试用
void vm_print(pgtbl_t pgtbl);

```

```

//获取页表项，alloc==true则分配一个新的物理页
pte_t* vm_get_pte(pgtbl_t pgtbl,uint64 va,bool alloc);
//映射函数，perm为权限位设置
void vm_map_pages(pgtbl_t pgtbl,uint64 va,uint64 pa,uint64 len,int perm);
//解映射，free==true则释放物理页
void vm_unmap_pages(pgtbl_t pgtbl,uint64 va,uint64 len,bool free);

//内核虚拟内存初始化，填充映射并修改stap寄存器
void kvm_init();
void kvm_hartinit();

```

虚拟内存这块需要参考上面的SV39的规范，进行地址转换和页表项的构建。

最后要将维护的内核页表地址写入**stap**寄存器中，这样机器才能进行地址的映射。

## 测试

使用 ENCUC-OSlab提供的vm\_print()函数，进行输出。

```

Enter start....
Enter main, cpuid: 0
-----Page table test -----
level-2 pgtbl: pa = 0x0000000080412000
.. level-1 pgtbl 0: pa = 0x0000000080413000
.. .. level-0 pgtbl 0: pa = 0x0000000080412000
.. .. .. physical page 0: pa = 0x0000000080800000 flags = 3
.. .. .. physical page 10: pa = 0x0000000080801000 flags = 7
.. .. level-0 pgtbl 1: pa = 0x0000000080412000
.. .. .. physical page 0: pa = 0x0000000080802000 flags = 11
.. level-1 pgtbl 1: pa = 0x0000000080416000
.. .. level-0 pgtbl 0: pa = 0x0000000080412000
.. .. .. physical page 0: pa = 0x0000000080802000 flags = 11
.. level-1 pgtbl 255: pa = 0x0000000080418000
.. .. level-0 pgtbl 511: pa = 0x0000000080412000
.. .. .. physical page 511: pa = 0x0000000080804000 flags = 5

```

可以看到，页表已经完成了初始化

## 4.trap

目前trap部分，跟做ECNU-OSlab3,代码部分已经基本完成

但实际上**测试不成功**

由于这里的 陷阱、中断等等与RISCV体系结构相关性比较大，我不是很懂

debug也难以进行，目前就是**没有实现**

## 第一阶段总结

从0到1实现操作系统并没有那么简单，就我这1个月的感觉来说，所做的与操作系统课上的教学相关性不强。唯一有点关联性的就是**内存分配和管理这块**，其它都是在做与RISCV体系相关性比较大的东西。

由于本人学校也没有学过RISCV相关的知识，所以基本是从零开始，比如要了解基本的汇编、寄存器等，

还要学会接触qemu、makefile这样的以前未接触的东西，虽然不需要掌握多深，但多多少少有难度。

关于**SBI**，初期花费了将近3个多星期在找SBI相关的文档和使用方法，但都需要体系结构的知识，其本身的代码也不好看懂，最后也没有学会怎么使用SBI，无耐放弃，从头开始。

第一次参加操作系统的比赛，结果并不理想，整体的内核骨架都没有搭建起来，感觉难度很大，需要的不仅仅是操作系统的知识，还要各种各样的知识和能力要求较大。。。