# 开启mmu实现虚实地址映射 | TimerのBlog

yanglianoo.github.io/2023/09/06/开启mmu实现虚实地址映射

2023年9月6日

## 1. 内存映射代码修改

在实现timeros的内存映射机制 | TimerのBlog (yanglianoo.github.io)这篇博客中我们已经实现了虚拟地址到物理地址映射的函数，但是我在实际调试中发现了一些bug，这里做一些修改。

在`find_pte_create`函数中：有两处修改

```
PageTableEntry* find_pte_create(PageTable* pt,VirtPageNum vpn)
{

    // 拿到虚拟页号的三级索引，保存到idx数组中
    size_t idx[3];
    indexes(vpn, idx);
    //根节点
    PhysPageNum ppn = pt->root_ppn;
    //从根节点开始遍历，如果没有pte，就分配一页内存，然后创建一个
    for (int i = 0; i < 3; i++)
    {
        //拿到具体的页表项
        PageTableEntry* pte =  &get_pte_array(ppn)[idx[i]];
            if (i == 2) {
                return pte;
            }
        //如果此项页表为空
            if (!PageTableEntry_is_valid(pte)) {
                //分配一页物理内存
                PhysPageNum frame =  StackFrameAllocator_alloc(&FrameAllocatorImpl);
               //新建一个页表项
               *pte =  PageTableEntry_new(frame,PTE_V);
                //压入栈中
                // push(&pt->frames,frame.value);
            }
        //取出进入下级页表的物理页号
        ppn = PageTableEntry_ppn(pte);
    }
}
```

- 首先是`size_t idx[3]`，这里`idx`应该定义成数组，之前定义的是一个指针是错误的，这是我的小失误

- 然后是不需要对frame的压栈操作，修改了pte的结构定义，不要栈来保存frame

c

```
/* 定义页表
项 */
typedef
struct
{
    uint64_t
bits;
}PageTableEn
try;
```

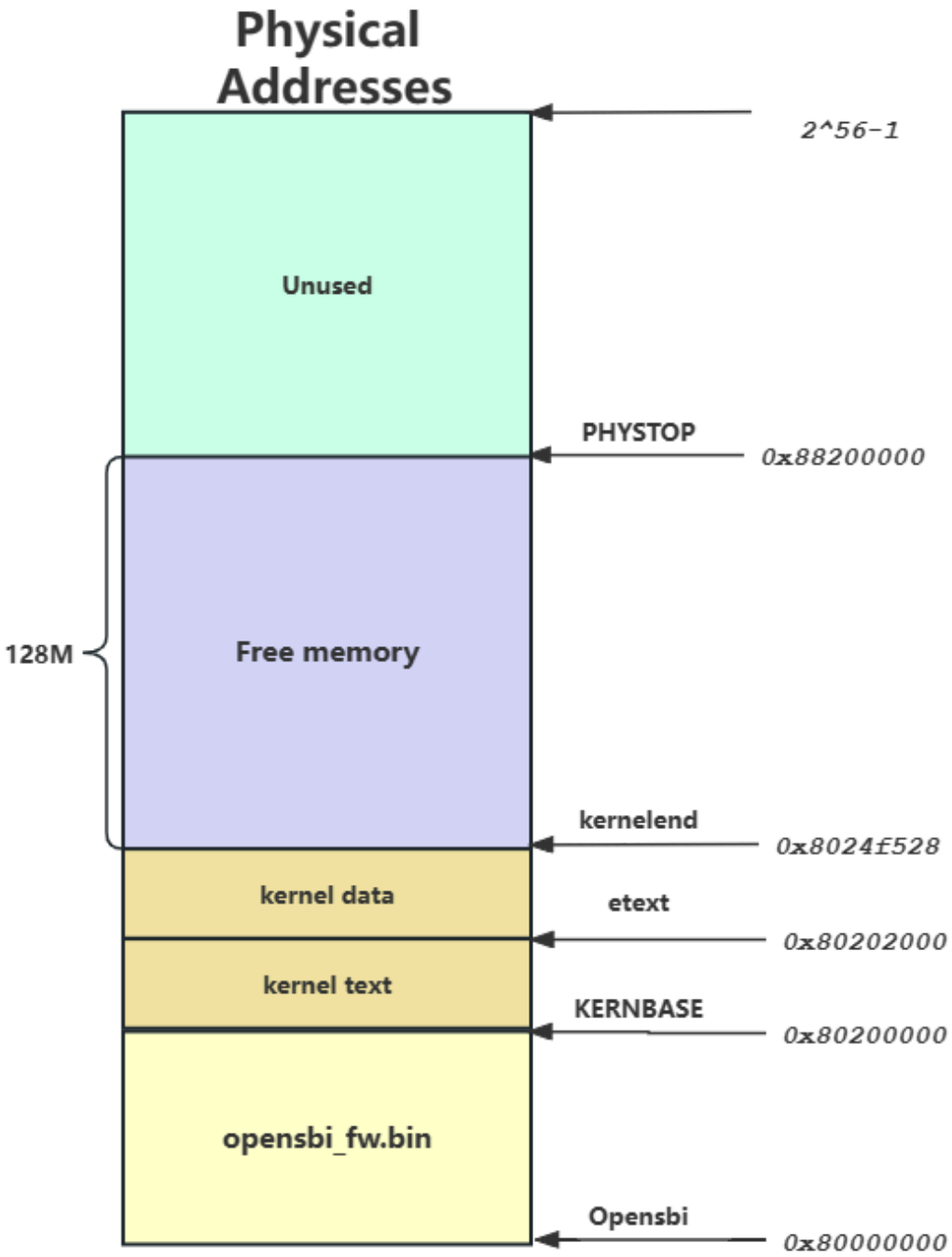
然后是PageTable_map函数的修改：新增了一个参数用于传递需要映射的内存长度，将从va开始的size大小的内存全部映射了，这里参考了xv6-riscv的映射实现。先计算需要映射多少页的内存，然后一页一页映射。

c

```
void PageTable_map(PageTable* pt,VirtAddr va, PhysAddr pa, u64 size ,uint8_t
pteflgs)
{
    if(size == 0)
        panic("mappages: size");

    PhysPageNum ppn = floor_phys(pa);
    VirtPageNum vpn = floor_virts(va);
    u64 last = (va.value + size - 1) / PAGE_SIZE;
    for(;;)
    {
        PageTableEntry* pte = find_pte_create(pt,vpn);
        assert(!PageTableEntry_is_valid(pte));
        *pte = PageTableEntry_new(ppn,PTE_V | pteflgs);

        if( vpn.value == last )
            break;

        // 一页一页映射
        vpn.value+=1;
        ppn.value+=1;
    }
}
```

## 2. 内存初始化

在开启虚拟地址之前，我们先来看一下现在内核的地址结构：




内核的起始地址是KERNBASE，内核的代码被编译器编译后是由代码段和数据段组成的，可以在os.map中看见各段的地址空间，代码段结束的地址设定为etext，数据段结束的地址设定为kernelend。然后指定从内核结束后向上128M的空间为空闲内存，可以给应用使用的。至于为什么etext和kernelend是上图那两个地址，我们来看一下os.ld文件，我做了一些修改：

os > os.ld

```
OUTPUT_ARCH(riscv)
ENTRY(_start)
BASE_ADDRESS = 0x80200000;
MEMORY
{
    ram (rxai!w) : ORIGIN = 0x80200000, LENGTH = 128M
}
SECTIONS
{
    skernel = .;    /* 定义内核起始内存地址 */
    .text : {
        *(.text .text.*)
    . = ALIGN(0x1000);
    PROVIDE(etext = .);
    } >ram

    .rodata : {
        *(.rodata .rodata.*)
    } >ram

    .data : {
        . = ALIGN(4096);
        *(.sdata .sdata.*)
        *(.data .data.*)
        PROVIDE(_data_end = .);
    } >ram

    .bss :{
        *(.sbss .sbss.*)
        *(.bss .bss.*)
        *(COMMON)
    } >ram

    PROVIDE(kernelend = .);
}
```

text结束的地址按页对齐

在链接脚本中，指定了代码段结束地址按页对齐，这是为了后续映射操作的方便性，因为我们映射的时候是按页进行映射的。

然后定义了两个符号：`PROVIDE(etext = .);`，`PROVIDE(kernelend = .);`，`etext`就代表了内核代码段结束的地址，`kernelend`就代表了内核结束的地址。这两个地址可以在`os.map`中找到，定义好符号后就可以用c语言去拿到值了

```
                0x0000000080202000                . = ALIGN (0x1000)
 *fill*         0x0000000080201d00        0x300
                0x0000000080202000                PROVIDE (etext = .)
```

```
 *(COMMON)
                0x000000008024f528                PROVIDE (kernelend = .)
OUTPUT(os.elf elf64-littleriscv)
```

在了解完毕内存分布之后，我们就可以来初始化内存了，我们可用的内存是从kernelend开始到PHYSTOP结束之间的大小，内核占据的代码段和数据段是不允许的，在address.c中来初始化可用内存：

c

```
StackFrameAllocator FrameAllocatorImpl;
extern char kernelend[];
void frame_alloctor_init()
{
    // 初始化时 kernelend 需向上取整
    StackFrameAllocator_new(&FrameAllocatorImpl);
    StackFrameAllocator_init(&FrameAllocatorImpl, \

ceil_phys(phys_addr_from_size_t(kernelend)), \

ceil_phys(phys_addr_from_size_t(PHYSTOP)));
    printk("Memoery start:%p\n",kernelend);
    printk("Memoery end:%p\n",PHYSTOP);
}
```

需要注意的是kernelend需要向上取整来对齐到0x80250000

## 3. 内存映射

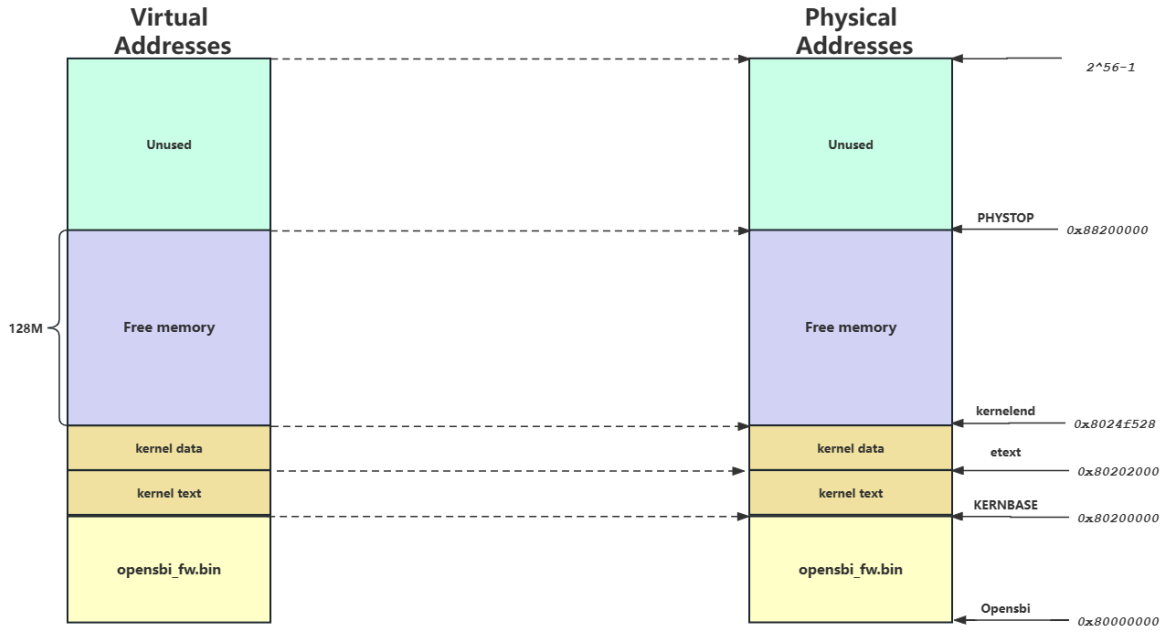
我们采用的内存映射方式为恒等映射，就是虚拟地址映射后的物理地址是相同的，这样在启用mmu后，原先的代码执行逻辑不变。在address中来进行恒等内存映射：

c

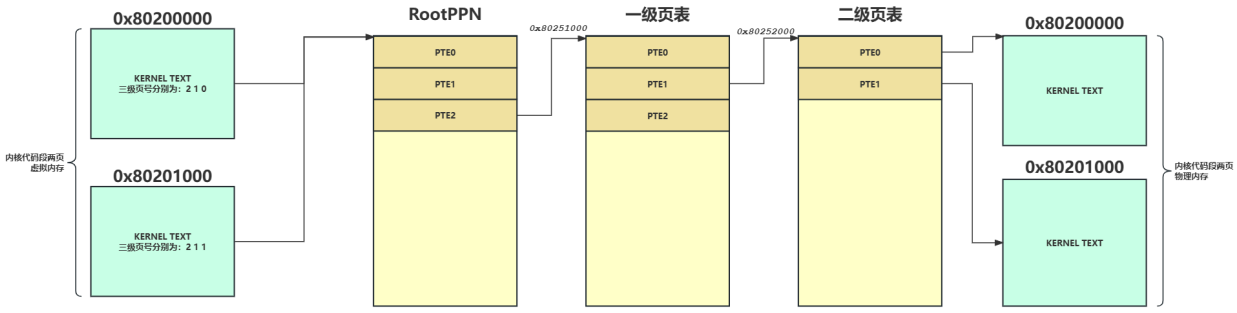
```
extern char etext[];
PageTable kvmmake(void)
{
    PageTable pt;
    PhysPageNum root_ppn =  StackFrameAllocator_alloc(&FrameAllocatorImpl);
    pt.root_ppn = root_ppn;
    printk("root_ppn:%p\n",phys_addr_from_phys_page_num(root_ppn));

    printk("etext:%p\n",(u64)etext);
    // map kernel text executable and read-only.
    PageTable_map(&pt,virt_addr_from_size_t(KERNBASE),phys_addr_from_size_t(KERNBASE),
\
                    (u64)etext-KERNBASE , PTE_R | PTE_X ) ;
    printk("finish kernel text map!\n");
    // map kernel data and the physical RAM we'll make use of.

PageTable_map(&pt,virt_addr_from_size_t((u64)etext),phys_addr_from_size_t((u64)etext ),
\
                    PHYSTOP - (u64)etext , PTE_R | PTE_W ) ;
    printk("finish kernel data and physical RAM map!\n");
    return pt;
}
PageTable kernel_pagetable;
void kvminit()
{
  kernel_pagetable = kvmmake();
}
```

首先建立一个根页表，从空闲内存中拿出一页，然后映射内核代码段，再映射数据段，代码段的属性是可执行可读的，数据段的属性是可读可写的，且U模式不可访问。由于我们现在是将U模式的应用和内核代码一起打包了，所以肯定U模式下的代码肯定是执行不了的，需要后面实现一个读取应用的模块来加载app。

映射完成后的内存长这样子：




然后是内核的映射表建立：内核的代码段只占两页内存： `0x80200000`,`0x80201000`，内核根页表放在`0x8025000`即空闲内存开始的第一页。虚拟地址`0x80200000`的三级页号的索引为 2 1 0，`0x80201000`的三级页号的索引为 2 1 1,通过下图的三次查表就对应上了具体的物理内存，要想彻底理解，还是自己手推一下映射关系。




## 4. 开启Sv39分页模式

要开启Sv39的分页模式，只需要去写`satp`的值就行了：设置为`Sv39`分页模式，然后将`root_ppn`的值写入。这里有一个刷新TLB的操作。

**快表** (TLB, Translation Lookaside Buffer)， 它维护了部分虚拟页号到页表项的键值对。当 MMU 进行地址转换的时候，首先会到快表中看看是否匹配，如果匹配的话直接取出页表项完成地址转换而无需访存；否则再去查页表并将键值对保存在快表中。一旦我们修改 satp 就会切换地址空间，快表中的键值对就会失效（因为快表保存着老地址空间的映射关系，切换到新地址空间后，老的映射关系就没用了）。为了确保 MMU 的地址转换能够及时与 satp 的修改同步，我们需要立即使用 `sfence.vma` 指令将快表清空，这样 MMU 就不会看到快表中已经过期的键值对了。

c

```
#define SATP_SV39 (8L << 60)
#define MAKE_SATP(pagetable) (SATP_SV39 | (((u64)pagetable)))

void kvminithart()
{
  // wait for any previous writes to the page table memory to
finish.
  sfence_vma();
  w_satp(MAKE_SATP(kernel_pagetable.root_ppn.value));
  // flush stale entries from the TLB.
  sfence_vma();
  reg_t satp = r_satp();
  printk("satp:%lx\n",satp);
}
```

sfence_vma和w_satp这两个函数定义在riscv.h中：

c

```
// supervisor address translation and
protection;
// holds the address of the page table.
static inline void w_satp(reg_t x)
{
  asm volatile("csrw satp, %0" : : "r"
(x));
}

static inline reg_t r_satp()
{
  reg_t x;
  asm volatile("csrr %0, satp" : "=r" (x)
);
  return x;
}

// 刷新 TLB.
static inline void sfence_vma()
{
  // the zero, zero means flush all TLB
entries.
  asm volatile("sfence.vma zero, zero");
}
```

如果内存正确映射的话，我们就可以看见打印satp寄存器的值了。

## 5. 测试

main.c

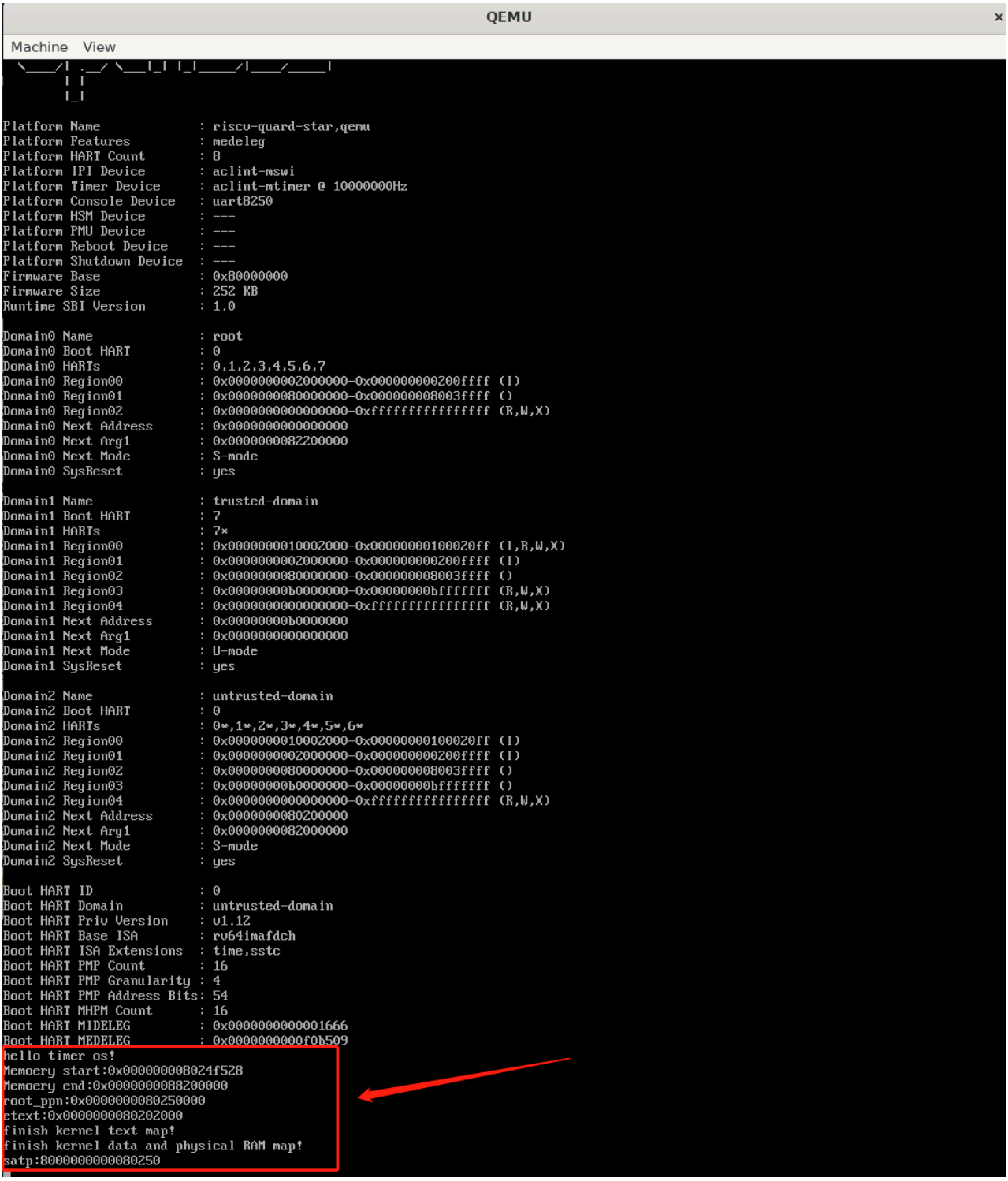
c

```
extern void
frame_alloctor_init();
extern void kvminit();
extern void kvminithart();
void os_main()
{
   printk("hello timer
os!\n");

   // 内存分配器初始化
   frame_alloctor_init();

   //初始化内存
   kvminit();

   //映射内核
   kvminithart();

   //trap初始化
   trap_init();


   while (1)
   {
      /* code */
   }

   // task_init();

   // timer_init();


   // run_first_task();

}
```

运行脚本测试：

sh

```
./bu
ild.
sh
./ru
n.sh
```




可以看见成功开启分页模式！！！！！

文章作者: Timer

文章链接: https://yanglianoo.github.io/2023/09/06/开启mmu实现虚实地址映射/



相关推荐