

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;
}PageTableEntry;
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
pte flgs)
{
    if(size == 0)
        panic("mappages: size");

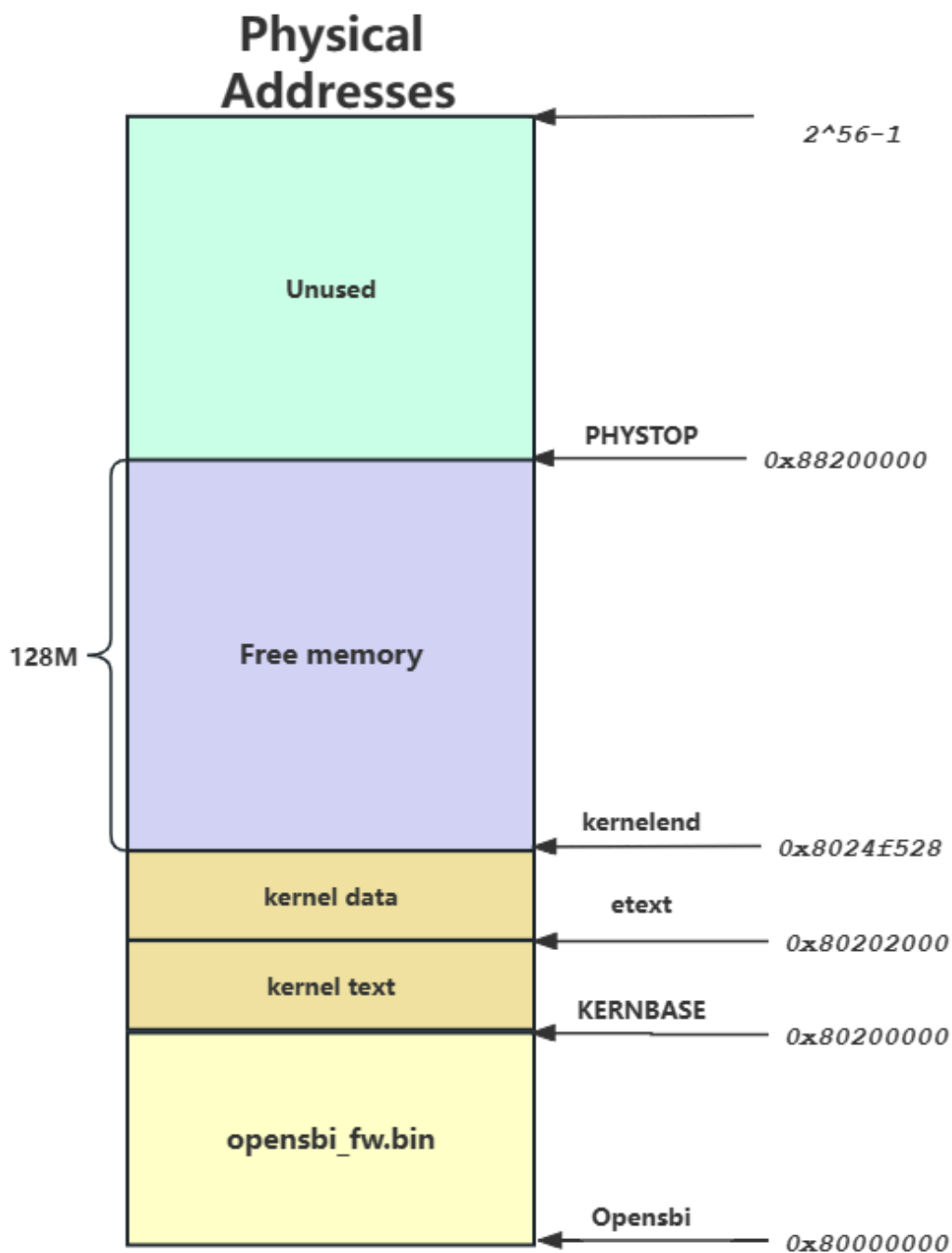
    PhysPageNum ppn = floor_phys(pa);
    VirtPageNum vpn = floor_virts(va);
    u64 last = (va.value + size - 1) / PAGE_SIZE;
    for(;;)
    {
        PageTableEntry* pte = find_pte_create(pt, vpn);
        assert(!PageTableEntry_is_valid(pte));
        *pte = PageTableEntry_new(ppn, PTE_V | pte flgs);

        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
1  OUTPUT_ARCH(riscv)
2  ENTRY(_start)
3  BASE_ADDRESS = 0x80200000;
4  MEMORY
5  {
6      ram (rxai!w) : ORIGIN = 0x80200000, LENGTH = 128M
7  }
8  SECTIONS
9  {
10     skernel = .; /* 定义内核起始内存地址 */
11     .text : {
12         *(.text .text.*)
13         . = ALIGN(0x1000);
14         PROVIDE(etext = .);
15     } >ram
16
17     .rodata : {
18         *(.rodata .rodata.*)
19     } >ram
20
21     .data : {
22         . = ALIGN(4096);
23         *(.sdata .sdata.*)
24         *(.data .data.*)
25         PROVIDE(_data_end = .);
26     } >ram
27
28     .bss :{
29         *(.sbss .sbss.*)
30         *(.bss .bss.*)
31         *(COMMON)
32     } >ram
33
34     PROVIDE(kernelend = .);
35 }

```

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

fill	0x0000000080202000	. = ALIGN (0x1000)
	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_allocator_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`中来进恒等内存映射：

```

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));

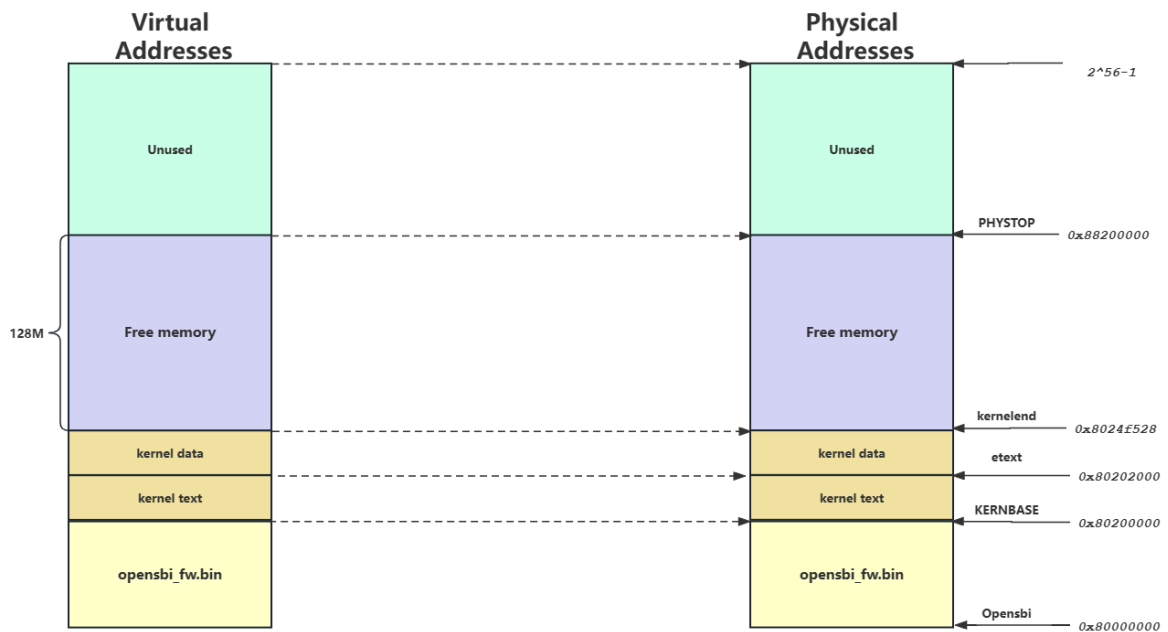
    printk("etext:%p\n",(u64)etext);
    // map kernel text executable and read-only.
    PageTable_map(&pt,virt_addr_from_size_t(KERNBASE),phys_addr_from_size_t(KERNBASE),
    \
        (u64)etext-KERNBASE , PTE_R | PTE_X );
    printk("finish kernel text map!\n");
    // map kernel data and the physical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

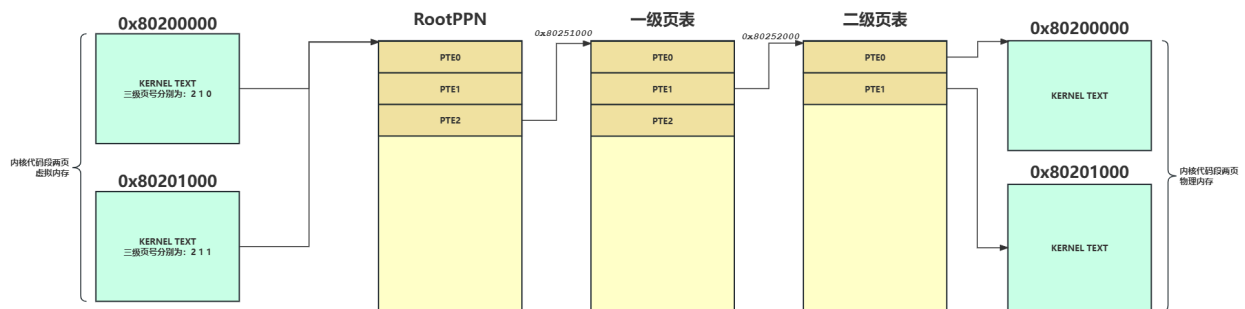
```

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 `0x80250000` 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 kvmminithart();
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
    kvmminithart();

    //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
```

```
QEMU
Machine View
Platform Name      : riscv-guard-star,qemu
Platform Features  : medeleg
Platform HART Count : 8
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 10000000Hz
Platform Console Device : uart8250
Platform HSM Device : ---
Platform PMU Device : ---
Platform Reboot Device : ---
Platform Shutdown Device : ---
Firmware Base      : 0x80000000
Firmware Size       : 252 KB
Runtime SBI Version : 1.0

Domain0 Name       : root
Domain0 Boot HART  : 0
Domain0 HARTs      : 0,1,2,3,4,5,6,7
Domain0 Region00   : 0x000000002000000-0x00000000200ffff (I)
Domain0 Region01   : 0x000000008000000-0x000000008003fff (I)
Domain0 Region02   : 0x000000000000000-0xfffffffffffff (R,W,X)
Domain0 Next Address : 0x000000000000000
Domain0 Next Arg1   : 0x0000000082200000
Domain0 Next Mode   : S-mode
Domain0 SysReset    : yes

Domain1 Name       : trusted-domain
Domain1 Boot HART  : 7
Domain1 HARTs      : 7*
Domain1 Region00   : 0x0000000010002000-0x00000000100020ff (I,R,W,X)
Domain1 Region01   : 0x0000000002000000-0x000000000200ffff (I)
Domain1 Region02   : 0x0000000008000000-0x0000000008003fff (I)
Domain1 Region03   : 0x000000000b000000-0x000000000bffffff (R,W,X)
Domain1 Region04   : 0x000000000000000-0xfffffffffffff (R,W,X)
Domain1 Next Address : 0x000000000b000000
Domain1 Next Arg1   : 0x000000000000000
Domain1 Next Mode   : U-mode
Domain1 SysReset    : yes

Domain2 Name       : untrusted-domain
Domain2 Boot HART  : 0
Domain2 HARTs      : 0*,1*,2*,3*,4*,5*,6*
Domain2 Region00   : 0x0000000010002000-0x00000000100020ff (I)
Domain2 Region01   : 0x0000000002000000-0x000000000200ffff (I)
Domain2 Region02   : 0x0000000008000000-0x0000000008003fff (I)
Domain2 Region03   : 0x000000000b000000-0x000000000bffffff (I)
Domain2 Region04   : 0x000000000000000-0xfffffffffffff (R,W,X)
Domain2 Next Address : 0x0000000008020000
Domain2 Next Arg1   : 0x0000000008200000
Domain2 Next Mode   : S-mode
Domain2 SysReset    : yes

Boot HART ID       : 0
Boot HART Domain   : untrusted-domain
Boot HART Priv Version : v1.12
Boot HART Base ISA : rv64imafdc
Boot HART ISA Extensions : time,sstc
Boot HART PMP Count : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 16
Boot HART MIDELEG   : 0x0000000000001666
Boot HART MEDELEG   : 0x0000000000f0b509

hello timer os!
Memory start:0x000000008024f528
Memory end:0x0000000080200000
root_ppn:0x0000000080250000
etext:0x0000000080202000
finish kernel text map!
finish kernel data and physical RAM map!
satp:800000000080250
```

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/>

版权声明: 本博客所有文章除特别声明外, 均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

相关推荐