实现timeros的内存映射机制 | TimerのBlog

yanglianoo.github.io/2023/08/31/实现timeros的内存映射机制

2023年8月31日

1. 页表操作

1.1 页表项定义

在上一篇博客中：用户态printf以及物理内存管理 | TimerのBlog (yanglianoo.github.io)已经展示了页表项的组成：

63	62 61	60 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

Sv39 page table entry.

页表项 (PTE, Page Table Entry)是一个`64bit`的数据，其中低10位存储的是下级物理页的属性的标志位；`10~54`这44位存储的是下级页表的物理页号。

具体定义如下：在address.h中

c

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

然后定义标志位：定义在address.h中

c

```c
// 定义位掩码常量
#define PTE_V (1 << 0)   //有效位
#define PTE_R (1 << 1)   //可读属
性
#define PTE_W (1 << 2)   //可写属
性
#define PTE_X (1 << 3)   //可执行
属性
#define PTE_U (1 << 4)   //用户访
问模式
#define PTE_G (1 << 5)   //全局映
射
#define PTE_A (1 << 6)   //访问标
志位
#define PTE_D (1 << 7)   //脏位
```

然后定义一些操作PTE的函数：在address.c中

```
/* 新建一个页表项 */
PageTableEntry PageTableEntry_new(PhysPageNum ppn, uint8_t
PTEFlags) {
    PageTableEntry entry;
    entry.bits = (ppn.value << 10) | PTEFlags;
    return entry;
}

/* 判断页表项是否为空 */
PageTableEntry PageTableEntry_empty() {
    PageTableEntry entry;
    entry.bits = 0;
    return entry;
}

/* 获取下级页表的物理页号 */
PhysPageNum PageTableEntry_ppn(PageTableEntry *entry) {
    PhysPageNum ppn;
    ppn.value = (entry->bits >> 10) & ((1ul << 44) - 1);
    return ppn;
}

/* 获取页表项的标志位 */
uint8_t PageTableEntry_flags(PageTableEntry *entry) {
    return entry->bits & 0xFF;
}

/* 判断页表项是否为空 */
bool PageTableEntry_is_valid(PageTableEntry *entry) {
    uint8_t entryFlags = PageTableEntry_flags(entry);
    return (entryFlags & PTE_V) != 0;
}
```

1.2 构造物理页帧的访问方法

假设我现在已经根据PTE拿到了物理页号，我要去访问此物理页号对应的物理帧的内存数据，因此需要定义了两个个辅助函数：

> 首先是以一个字节作为单位访问数据，拿到物理页号之后转换为对应的物理地址，得到的物理地址是此物理帧的开头，将其转换为一个`uint8_t*`类型的指针，这样根据此指针就可操作这一页的4096个字节了

C

```
uint8_t* get_bytes_array(PhysPageNum ppn)
{
    // 先从物理页号转换为物理地址
    PhysAddr addr =
phys_addr_from_phys_page_num(ppn);
    return (uint8_t*) addr.value;
}
```

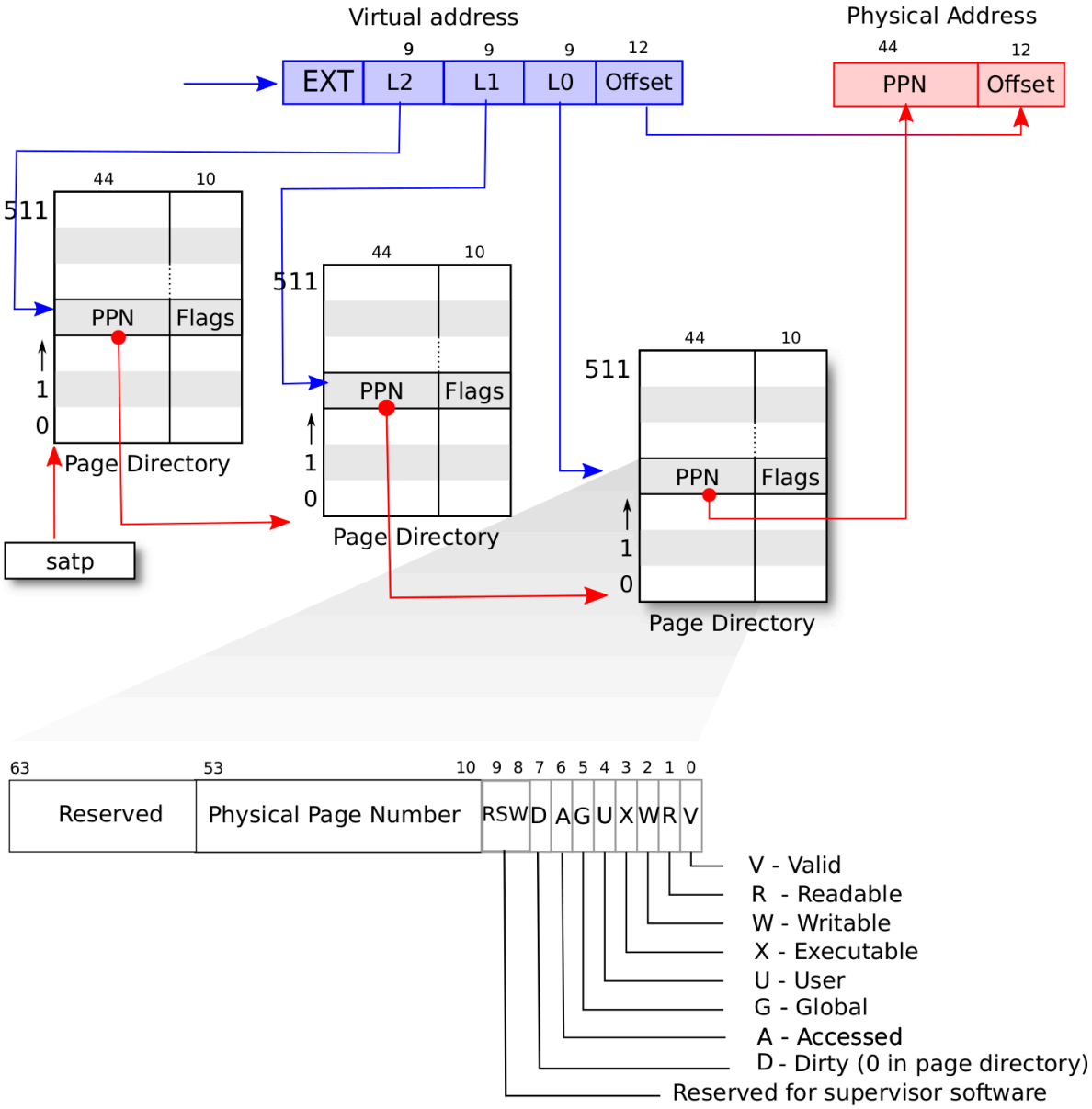
然后假设物理页帧中存储的是一个个PTE，我现在需要去访问这一个个PTE。同样需要先拿到物理页号之后转换为对应的物理地址，然后将其转换为`PageTableEntry*`的指针就可以了

C

```
PageTableEntry* get_pte_array(PhysPageNum ppn)
{
    // 先从物理页号转换为物理地址
    PhysAddr addr =
phys_addr_from_phys_page_num(ppn);
    return (PageTableEntry*) addr.value;
}
```

2.虚实地址映射

2.1 rCore的内存映射实现



MMU在寻址时的流程如上图，先要拿到`Virtual address`的三级页号索引，定义如下的辅助函数：

C

```
/* 拿到虚拟页号的三级索引，按照从高到低的顺序返回 */
void indexes(VirtPageNum vpn, size_t* result)
{
    size_t idx[3];
    for (int i = 2; i >= 0; i--) {
        idx[i] = vpn.value & 0x1ff;   // 1_1111_1111 =
0x1ff
        vpn.value >>= 9;
    }

    for (int i = 0; i < 3; i++) {
        result[i] = idx[i];
    }
}
```

然后我们需要根据拿到的虚拟地址的三级页号索引来查找页表项，当然在最开始的时候多级页表的页表项里是没有数据的，这就需要我们去进行填充。

在这之前需要定义一个管理页表的结构体：每个应用的地址空间都对应一个不同的多级页表，这也就意味这不同页表的起始地址（即页表根节点的地址）是不一样的。因此 `PageTable` 要保存它根节点的物理页号 `root_ppn` 作为页表唯一的区分标志。此外，向量 `frames`保存了页表所有的节点（包括根节点）所在的物理页帧。

C

```
/* 定义页表 */
typedef struct {
    PhysPageNum root_ppn; //
根节点
    Stack frames;         //
页帧
}PageTable;
```

然后在来看查找页表项，填充页表项的操作：传入一个`PageTable`，根据此页表的根节点开始遍历，根节点的物理页号是保存在`satp`寄存器中的，从页表中根据虚拟地址的页表项索引来取出具体的页表项，如果此页表项为空，则分配一页内存，然后新建一个页表项进行填充。直到三级页表索引完毕，会返回虚拟地址最终对应的三级页表的页表项，此时三级页表的页表项是空的，在进行map时只需要对此页表项赋值就行。

C

```
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;
    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);
    }

}
```

C

```
PageTableEntry* find_pte(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;
    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)) {
                return NULL;
            }
        //取出进入下级页表的物理页号
        ppn = PageTableEntry_ppn(pte);
    }

}
```

`find_pte_create` 和`find_pte`基本一样，区别在于`find_pte`只会去查找页表项不会去创建。

有了上面这个函数，我们就可以建立虚实隐射关系了：只需要将你需要映射的物理页号与虚拟地址索引的三级页表中的页表项对应起来即可

C

```
void PageTable_map(PageTable* pt,VirtPageNum vpn, PhysPageNum ppn, uint8_t
pteflgs)
{
    PageTableEntry* pte = find_pte_create(pt,vpn);
    assert(!PageTableEntry_is_valid(pte));
    *pte = PageTableEntry_new(ppn,PTE_V | pteflgs);
}
```

然后还需一个解除映射的函数：先查找页表项，如果找到了就将其设为空就行了。

C

```
void PageTable_unmap(PageTable* pt,
VirtPageNum vpn)
{
    PageTableEntry* pte = find_pte(pt,vpn);
    assert(!PageTableEntry_is_valid(pte));
    *pte = PageTableEntry_empty();
}
```

> 按逻辑上来说这里如果解除映射之后，应该需要释放掉对应的页内存才对

上面用到了一个assert函数，这是断言的宏，具体的实现如下，新增了一个assert.c和assert.h

C

```
#ifndef TOS_ASSERT_H
#define TOS_ASSERT_H

void assertion_failure(char *exp, char *file, char *base, int line);


/*
__FILE__是一个预定义的宏，在C语言中表示当前源文件的文件名。
在预处理阶段，编译器会将所有的__FILE__宏替换为当前源文件的文件名字符串。
__BASE_FILE__是一个预定义的宏，在某些编译器中用于表示当前编译单元的顶层源文件的文
件名，
即当前源文件所属的工程或者库的主文件名。
__LINE__是一个预定义的宏，在C语言中表示当前代码所在的行号。
在预处理阶段，编译器会将所有的__LINE__宏替换为当前代码所在的行号。
*/


#define assert(exp) \
    if (exp)        \
        ;           \
    else            \
        assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)

#endif
```

c

```c
#include <timeros/assert.h>
#include <timeros/types.h>

// 强制阻塞
static void spin(char *name)
{
    printk("spinning in %s ...\n", name);
    while (true)
        ;
}


//提示报错信息
void assertion_failure(char *exp, char *file, char *base, int
line)
{
    printk(
        "\n--> assert(%s) failed!!!\n"
        "--> file: %s \n"
        "--> base: %s \n"
        "--> line: %d \n",
        exp, file, base, line);

    spin("assertion_failure()");
}
```

2.2 xv6的内存映射实现

xv6的内存映射代码实现在这里：xv6-riscv/kernel/vm.c at riscv · mit-pdos/xv6-riscv (github.com)

> 首先来看第一个函数`kvmmake`，在这个函数里先创建了一个根页表，为其分配一页内存，然后调用`kvmmap`函数来进行内存映射。这里建立的是恒等映射关系，把`PHYSTOP`以下的内存全部映射了，使用一个多级页表将内核的内存全部映射了。

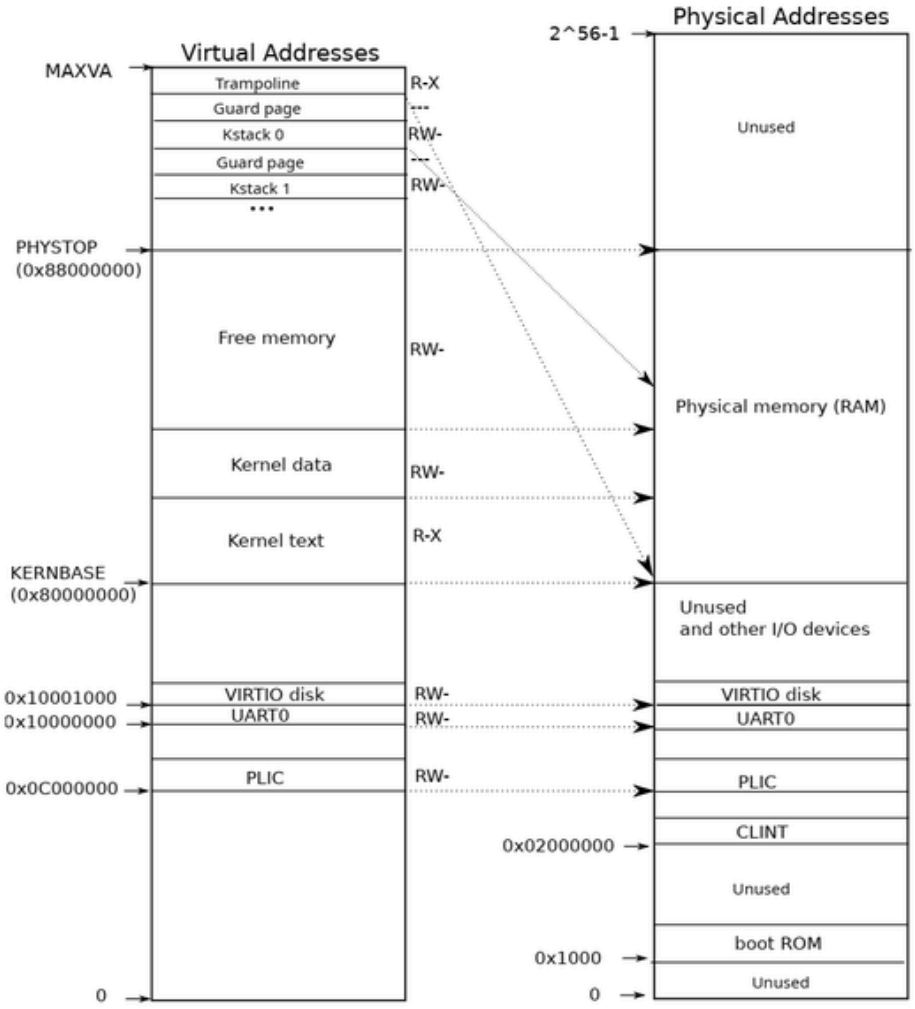


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

C

```
pagetable_t
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // uart registers
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  kvmmap(kpgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  c(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  // allocate and map a kernel stack for each process.
  proc_mapstacks(kpgtbl);

  return kpgtbl;
}
```

然后我们来看一下kvmmap的实现，它实际上会去调mappages函数

C

```
// add a mapping to the kernel page table.
// only used when booting.
// does not flush TLB or enable paging.
void
kvmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int
perm)
{
  if(mappages(kpgtbl, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
```

继续看mappages函数，此函数有五个参数，分别是：三级页表地址，需要建立映射的虚拟地址和物理地址，映射的内存大小，映射的内存的属性。这个函数干的事情就是根据虚拟地址遍历三级页表，如果没有映射就新建页表项填充，最后返回一级页表的页表项指针，最后对其赋值就完成了映射操作，这个函数就对应着在上面rCore中实现的map函数的逻辑，区别在于这里由于传入了一个映射内存的长度，所以在映射完一页内存后，虚拟地址和需要映射的物理内存地址都将向上增长一页继续映射，直到全部映射完成。查找和建立页表项的操作是在walk函数中完成的

c

```
int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  if(size == 0) //如果映射的大小为0，则panic
    panic("mappages: size");

  /* 按页对齐 */
  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);

  for(;;){
    //遍历三级页表，分配物理页帧，建立映射关系，第一级页表中指定虚拟地址 va 对应的页表项的指针
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    // 如果已经被映射过了，则panic
    if(*pte & PTE_V)
      panic("mappages: remap");
    //设置pte
    *pte = PA2PTE(pa) | perm | PTE_V;

    if(a == last)
      break;
    // 一页一页映射
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

walk函数其实就对应着上面实现的find_pte_create函数，实现逻辑一摸一样

c

```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      //如果已经映射到一个下一级页表，则将 pagetable 更新为下一级页表的物理
地址。
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  //返回第一级页表中指定虚拟地址 va 对应的页表项的指针
  return &pagetable[PX(0, va)];
}
```

参考链接

- xv6–内存管理_panic remap_lhw—9999的博客-CSDN博客

- 管理 SV39 多级页表 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)