Lab3实验报告

变量/宏/函数定义说明

- sv39中采用的是三级页表，从高到低三级页表的页码分别称作PDX1, PDX0和PTX(Page Table Index)。
- 39位的虚拟地址， `PDX1(9)|PDX0(9)|PTX(9)|PGOFF(12)` ，相关的宏定义通过位运算实现
- `PPN=PDX1(9)|PDX0(9)|PTX(9)`
- 虚拟地址通过 `PGADDR(PDX(la), PTX(la), PGOFF(la))` 得到
- Sv39的物理地址： `PPN2(26)|PPN1(9)|PPN0(9)|PGOFF(12)`
- Sv39的页表项： `PPN2(26) | PPN1(9) | PPN0(9) |Reserved(2)|D|A|G|U|X|W|R|V|`

结构体定义

- 连续虚拟内存区域：描述一段从vm_start到vm_end的连续虚拟内存。通过包含一个list_entry_t成员，把同一个页表对应的多个vma_struct结构体串成一个链表，在链表里把它们按照区间的起始点进行排序

```
struct vma_struct {
    // 关联的上层内存管理器
    struct mm_struct *vm_mm;
    // 描述的虚拟内存的起始地址
    uintptr_t vm_start;
    // 描述的虚拟内存的截止地址
    uintptr_t vm_end;
    // 当前虚拟内存块的属性flags（读/写/可执行）
    uint_t vm_flags;
    // 连续虚拟内存块链表节点（mm_struct->mmap_list）
    list_entry_t list_link;  // 用于串成链表
};
```

- 内存管理结构体：把一个页表对应的vma_struct信息组合起来，包括vma_struct链表的首指针，对应的页表在内存里的指针，vma_struct链表的元素个数。

```
struct mm_struct {
    // 连续虚拟内存块链表（内部节点虚拟内存块的起始、截止地址必须全局有序，且不能出现重叠）
    list_entry_t mmap_list;
    // 当前访问的mmap_list链表中的vma块(由于局部性原理，之前访问过的vma有更大可能会在后
    // 续继续访问，该缓存可以减少从mmap_list中进行遍历查找的次数，提高效率）
    struct vma_struct *mmap_cache;
    // 当前mm_struct关联的一级页表的指针
    pde_t *pgdir;
    // 当前mm_struct->mmap_list中vma块的数量
    int map_count;
    // 用于虚拟内存置换算法的属性，使用void*指针做到通用
    // lab中默认的swap_fifo替换算法中，将其做为了一个先进先出链表队列
    void *sm_priv;
};
```

练习2：深入理解不同分页模式的工作原理（思考题）

首先，我们对get_pte()函数做一个解析。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {}
```

参数部分：

1. pgdir:内核中PDT的虚拟地址
2. la:需要被映射的虚拟地址
3. create:是否准许分配一个页面
4. 返回虚拟地址的页表项

```
// 把虚拟地址左移30位得到PTX1的索引，pgdir为页表基址，&pgdir[PDX1(la)]则为页表中第PDX1项的起始地址，让
   pde_t *pdep1 = &pgdir[PDX1(la)];            // 找到对应的Giga Page
   if (!(*pdep1 & PTE_V)) {                    // 如果下一级页表不存在，那就给它分配一页，创造新页表
       struct Page *page;
       if (!create || (page = alloc_page()) == NULL) { // create为1并且成功找到多的1个物理页才分配
           return NULL;
       }
       set_page_ref(page, 1);                          // 增加对物理页面page的引用数
       uintptr_t pa = page2pa(page);                   // page2pa 将一个页转换成这个页的物理地址
       memset(KADDR(pa), 0, PGSIZE);                   // 把物理地址pa映射到虚拟地址中，之后的一页内
       // 不管页表怎么构造，我们确保物理地址和虚拟地址的偏移量始终相同，那么就可以用这种方式完成对物理内
       *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
   }
```

第一段是根据 la 前9位的索引，让 pdep1 指向 PTX1 顶级页表的第 PDX1(la) 项，如果该项不存在并且设置为可以被分配，那就分配一个物理页并且增加1次引用。
对于 page2pa 函数，在此一并分析。函数定义如下：

```
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }
// 页块数组中，页块的指针-页块数组的起始地址为页块编号，再把编号加上基准页数即为整个物理内存分页后的页编号
// 物理页码（PPN）只是一个数字，它表示页在物理内存中的索引或顺序位置。

static inline uintptr_t page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
    // 物理页码左移12位后，就得到了以改页码
}
```

page2pa 的功能是把页块结构体指针 page 映射到对应的物理页的起始地址上。其中调用了 page2ppn 这个函数。由于在Lab2中，在内核代码结束的内存空间里，我们用一个页块数组来存储从物理地址 0x80000000-0x88000000 的地址空间按照4KB分页后的页数。 pages 是该数组的起始地址， page-pages 即为数组内的偏移，加上 nbase 即为对整个物理内存分页后的物理页码，左移12位号即为该页的起始物理地址。

继续分析 get_pte() ，pa 中存储 page 页的物理地址，之后调用 memset() 将pa对应的虚拟地址上的一页内容全置为0，之后填入页表项内容（将物理地址和虚拟地址建立映射）。对于第二段，跟第一段的逻辑类似，来构造下一级页表 PDX0 的页表项，根据构造号的页表项再来索引第三级页表 PTX ，并最终返回 PTX 页表项的虚拟地址，在此不再赘述。

思考题：

- get_pte()函数中有两段形式类似的代码， 结合sv32，sv39，sv48的异同，解释这两段代码为什么如此相像。

从上面的分析中可以看到，在 get_pte() 函数中，两段形式类似的代码实际上对应了 sv39 页表模式中的两级页表访问。第一段代码PDX1中，根据la的前9位索引导到页表项，访问或分配 PDX0 的页表项。在此过程中，如果页表项不存在，那么需要一个物理页面来存放下一级页表，所以需要分配一个物理页并将其映射到对应的物理地址上，最终把这个物理地址存到页表项中；第二段代码访问或分配了 PTX 的页表项，进行的操作类似。两段代码看起来相似，因为它们执行的是类似的操作，但在不同级别的页表上。因为是从顶级页表开始， get_pte() 只需处理两级，并向下查找直到找到或创建所需的页表项。这种模式是多级页表系统的一种常见模式，无论是采用二级页表的 sv32 、采用三级页表的 sv39 、还是采用四级页表的 sv48 ，都遵循相似的逻辑，只是层数和索引方式有所不同。

- 目前get_pte()函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？

我认为这种写法没问题，也没有必要将两个操作分开。理由如下：

- 简化代码。只需要调用一个函数，使得代码更加简洁。同时find和alloc两个操作都需要对PDX0和PDX1进行索引，拆开写会使得代码冗余，还需要增加新的函数参数和返回值，不利于维护。
- 保持一致性。不单独提供alloc的函数接口，将查找和分配页表项的操作原子化，避免在其他代码中错误地调用了分配的函数，减少错误。

练习3：给未被映射的地址映射上物理页（需要编程）

`do_fault` 函数的基本处理逻辑：

- `do_pgfault` 函数接收中断错误码和引起错误的线性地址，还接受了 `mm_struct` 结构，用以访问页表。
- `do_pgfault` 对错误码进行了处理，判断其究竟是否是因为缺页造成的页访问异常,还是因为非法的虚拟地址访问。

> 在 `vmm_struct` 寻找包裹触发异常的地址的 `vma_struct` ,如果没有找到则说明是非法的地址访问，返回 `-E_INVAL` 错误码。
> 如果成功找到，则说明访问的是合法的虚拟地址，之后进行下一步操作。

- 对于合法的访问，进一步找到引起异常的地址所对应的PTX页表项，判断是该页是真的不存在，还是被交换到了磁盘中。

> 如果pte为0，则说明该页的确不存在，需要为其分配一个初始化后全新的物理页，并建立映射虚实关系。
> 如果pte不为0，且开启了swap磁盘虚拟内存交换机制，则说明该页在磁盘中，而我们填写的代码需要完成从磁盘中拿回该页的功能。

- 如果do_pafault实现正确，那么此时将能够正确地访问到虚拟地址对应的物理页，程序正常往下执行。

下面给出从磁盘拿回缺页的代码：

```
else {
        if (swap_init_ok) {// 开启了swap磁盘虚拟内存交换机制
            struct Page *page = NULL;
            // 将物理页换入到内存中
            // 将addr线性地址对应的物理页数据从磁盘交换到物理内存中(令Page指针指向交换成功后的物理页)
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                // swap_in返回值不为0，表示换入失败
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            // 将交换进来的page页与mm->padir页表中对应addr的二级页表项建立映射关系(perm标识这个二级页
            page_insert(mm->pgdir, page, addr, perm);
            // 当前page是为可交换的，将其加入全局虚拟内存交换管理器的管理
            swap_map_swappable(mm, addr, page, 1);
            page->pra_vaddr = addr;
        } else {
            // 如果没有开启swap磁盘虚拟内存交换机制，但是却执行至此，则出现了问题
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
    }
```

代码流程：

1. 调用 `swap_in()` ,将页面换入Page对应的一页物理内存中。
2. 将page页和 `mm->pgdir` (注意这里不是 `boot_pgdir`)页表对应addr的PTX页表项建立映射关系，同时权限位设为perm。
3. 设置当前page为可交换的，将其加入全局虚拟内存交换管理器的管理。

> 细节点：在这里，实际并没有考虑是否有空闲的物理内存页，调用 `swap_in` 的时候， `swap_in` 会调用 `alloc_page` 来
> 分配一个物理页，如果没有足够的内存页，这时 `alloc_page` 才会采用页面置换算法找到一个“受害者”，将其换到磁盘上。

问题回答：

- 请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处。

PDE和PTE中，除了直接映射到下一级页表或者物理地址的索引，还有一些特殊的位标志，即低八位 `|D|A|G|U|X|W|R|V|` ，分别代表

1. D (Dirty)：脏位。如果该页自上次被清除以来已经被写过，则该位被设置。这对于决定是否需要将页面内容写回到磁盘非常重要。
2. A (Accessed)：访问位。如果该页已被读取或写入，则设置此位。这可以用于实现某些页替换算法，比如最近最少使用（LRU）算法。
3. G (Global)：全局位。如果设置了此位，这个页表项会被所有的地址空间所共享，而不是只属于一个特定的地址空间。
4. U (User)：用户位。如果设置了此位，这表示这个页面可以被用户模式的代码访问。如果没设置，只有内核模式的代码才能访问。
5. X (Executable)：可执行位。如果设置了此位，这表示这个页面上的内容可以被执行。
6. W (Writeable)：可写位。如果设置了此位，这表示这个页面可以被写入。
7. R (Readable)：可读位。如果设置了此位，这表示这个页面可以被读取。
8. V (Valid)：有效位。如果设置了此位，这表示这个页表项是有效的。如果没设置，表示这个页表项不指向一个有效的物理页面。

在实现页面替换算法时，通过PTE和PDE，可以设置换入页面的访问权限和页属性等，不需要单独用数据结构或者算法来完成权限和属性的替换。

- 如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

1. 记录异常信息，硬件记录引发二次异常的地址。
2. 保存当前状态：硬件需要保存当前的执行状态，包括程序计数器、寄存器状态等上下文信息。
3. 触发异常处理程序：硬件将控制权交给操作系统的异常处理程序。执行流会跳转到操作系统定义的另一个异常处理例程。
4. 等待操作系统响应，处理后跳转到引发二次异常的地址，继续执行原来的缺页服务例程。

- 数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系？如果有，其对应关系是啥?

Page数组，用来表示物理内存中的所有页面（0x80000000-0x88000000）。Page的每一项对应于物理内存中的一个页面，与页表中的页表项和页目录项有关系，关系如下：

> 某一项的Page结构体的指针-起始的Page结构体数组的基址，可以得到其对应的物理页的页码，加上0x80000000/4096KB,即为整个地址空间按页分割后的物理页码，左移12位后即为该Page对应页的起始物理地址。所以，每一项Page都与一个物理页面一一对应。
> 而PTE包含一个指向物理内存中某个页面的指针。因此，可以说每个PTE直接对应于Page数组中的每一项结构体。它们是一一对应的。
> PDE指向一个页表。对于PDX0页表的页目录项，对应的PTX页表有512个页表项，每个PTE又对应于Page数组的一项。因此PDX0页表的页目录项对应512个Page结构体，而PDX1的页目录项则对应512*512个Page结构体。

练习4：补充完成Clock页替换算法（需要编程）

补充的代码如下：

```
list_entry_t pra_list_head, *curr_ptr;
static int
_clock_init_mm(struct mm_struct *mm)
{
     // 初始化pra_list_head为空链表
     list_init(&pra_list_head);
     // 初始化当前指针curr_ptr指向pra_list_head，表示当前页面替换位置为链表头
     curr_ptr = &pra_list_head;
     // 将mm的私有成员指针指向pra_list_head，用于后续的页面替换算法操作
     mm->sm_priv = &pra_list_head;
     return 0;
}
```

clock置换算法的初始化函数与fifo置换算法的初始化函数类似，对链表进行初始化，同时用一个curr_ptr记录链表头。

```
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
{
    list_entry_t *entry=&(page->pra_page_link);
    assert(entry != NULL && curr_ptr != NULL);
    // 将页面page插入到页面链表pra_list_head的末尾
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_add_before(head,entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited = 1;
    return 0;
}
```

与fifo类似，将新加入的page放在页面链表末尾，由于是一个双向链表，所以这里通过 `list_add_before` 操作即可，与fifo不同的是，这里需要设置该页的visited为1，表示刚刚被访问。这一点十分重要，关系到下面实现置换算法成功与否。

```
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
        assert(head != NULL);
    assert(in_tick==0);
    while (1) {
        // 遍历页面链表pra_list_head，查找最早未被访问的页面
        // 获取当前页面对应的Page结构指针
        // 如果当前页面未被访问，则将该页面从页面链表中删除，并将该页面指针赋值给ptr_page作为换出页面
        // 如果当前页面已被访问，则将visited标志置为0，表示该页面已被重新访问
        if(curr_ptr == head){
            curr_ptr = list_next(curr_ptr);
            continue;
        }
        struct Page * page = le2page(curr_ptr,pra_page_link);
        list_entry_t* next_entry = list_next(curr_ptr);
        if(page->visited == 0){
            cprintf("curr_ptr 0xffffffff%x\n",curr_ptr);
            list_del(curr_ptr);// 将该页面从页面链表中删除
            *ptr_page = page;
            curr_ptr = next_entry;
            break;
        }
        else{
            page->visited = 0;
            curr_ptr = list_next(curr_ptr);
        }
    }
    return 0;
}
```

> 在该算法中，我们用 curr_ptr 这个全局指针来记录当前遍历到哪一个Page页上。
> clock页替换算法与fifo不同的地方在于决策被置换的页面（受害者），也就是以上函数。在该函数中，我们按照理论课讲解时间片算法的步骤，通过while循环持续遍历循环链表，如果遍历到头结点则跳过（头结点为空）。对于每一个页面，如果其标志位visiter为1，则将其置为0；如果标志位为0，则将该页面从页面链表中删除。ptr_page中存储换出页面的结构体指针，退出循环，函数返回。
> 注意点：按照算法流程，找到一个“受害者”并确定将其换出后，时间片要再向前推进一次，这里用next_entry来提前记录，成功找到后就把next_entry赋给curr_ptr，而下一次调用该函数时，curr_ptr就会此位置继续遍历。

- 比较Clock页替换算法和FIFO算法的不同。
  不同的地方主要在于_clock_swap_out_victim函数，寻找被换出的“受害者”。

- FIFO实现较为容易，由于本身采用链表来维护所有的物理页，只需要每次将新页放到链表尾部，每次置换链表头部对应的页面即可
- Clock实现稍微复杂一些，需要遍历循环链表，来修改和访问每个页的visited位。