虚拟内存管理

页表机制和中断异常处理机制

提供一个比实际物理内存空间更大的虚拟内存空间

虚存管理总体框架

- 完成初始化虚拟内存管理机制：IDE硬盘读写，缺页异常处理
- 设置虚拟页空间和物理页帧空间，表述不在物理内存中的虚拟页
- 完成建立页表映射、页访问异常处理操作等函数实现
- 执行访存测试，查看建立的页表项是否能够正确完成虚实地址映射
- 执行访存测试，查看是否正确描述了虚拟内存页在物理内存中还是在硬盘上
- 执行访存测试，查看是否能够正确把虚拟内存在物理内存和硬盘之间进行传递
- 执行访存测试，查看是否正确实现了页面替换算法

(1)

mm_struct和vma_strut表述不在物理内存中的合法虚拟页，当ucore访问合法虚拟页时，由于没有虚实地址映射会产生页访问异常。do_pgfault函数会申请一个空闲物理页并建立虚实映射关系。使得合法虚拟页有实际的物理页帧对应

(2)

ide_init

页面置换算法的实现存在对硬盘数据块的读写，ide_init完成对用于页换入换出的硬盘swap的初始化工作

(3)

引起page fault

- 虚拟地址内存访问越界
- 写只读页的非法地址访问
- 数据被临时换到磁盘上
- 没有分配内存的合法地址访问

(4)

find_vma的逻辑很简单

```
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
                bool found = 0;
                list_entry_t *list = &(mm->mmap_list), *le = list;
                while ((le = list_next(le)) != list) {
                    vma = le2vma(le, list_link);
                    if (vma->vm_start<=addr && addr < vma->vm_end) {
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    vma = NULL;
                }
        }
        if (vma != NULL) {
            mm->mmap_cache = vma;
        }
    }
    return vma;
}
```

值得一提的是其中的`vma=mm->mmap_cache`,mmap_cache指向当前正在使用的虚拟内存空间，由于操作系统的局部性原理，当前正在用到的虚拟内存空间再接下来的操作中还会用到，直接将mmap_cache给vma，有时就不需要查链表，即不进入下面的循环。使得mm_strcut数据结构的查询加速30%以上

(5)

insert_vma_struct的位置问题

(6)

kmalloc分配地址空间，显然需要考虑地址空间大小小于一个页大小的情况，这里采用 加上一个PGSIZE-1 的方法进行补齐，这个技巧在向上取整`ROUND_UP`的时候也有用到

```
void *
kmalloc(size_t n) {
    void * ptr=NULL;
    struct Page *base=NULL;
    assert(n > 0 && n < 1024*0124);
    int num_pages=(n+PGSIZE-1)/PGSIZE;
    base = alloc_pages(num_pages);
```

```
    assert(base != NULL);
    ptr=page2kva(base);
    return ptr;
}
```

(7)

再次考虑mm_strcut 和vma_struct,实验二中有关内存的数据结构和相关操作都是直接针对实际存在的资源---物理内存空间的管理，没有从应用程序的角度考虑。通过这两个数据结构，当内存产生page fault异常时，可获得访问内存的方式以及具体的虚拟内存空间，体现在do_pgfault中的

```
struct vma_struct *vma=find_vma(mm,addr)
...
error_code
```

通过判断，进入读页表项阶段

- 如果页表项为全0，说明是还没分配物理页的合法空间，调用`pgdir_alloc_page`进行分配`alloc_page`、映射`page_insert`和可换出化`swap_map_swappable`
- 如果不为0(高24位表示在硬盘起始扇区的位置)，需要进行换入`swap_in`、映射`page_insert`和可换出化`swap_map_swappable`

(8)

换入的过程设计到读磁盘`swapfs_read`,自然需要用到表项的高24位，所以需要先进行`get_pte`，而由于读出来的是一张真实的页，还需要提前分配一个物理页result，如果分配失败，则进入`swap_out`,最终将result传给`swap_in`的参数ptr_result完成换入

(9)

page_insert主要目的是修改表项和相关的参数，根据页目录表和线性地址找到相应的表项，增加page的被引用数，如果该表项存在物理页对应，需要检查新的物理页和原来的物理页是否一致，若一致，将刚刚增加的引用数减一，若不一致删除该表项，最终都要重新写表项，包括物理地址、存在位和perm,然后更新TLB

(10)

swap_map_swappable的调用通过了一个manager这样方便后续更改替换算法，而实现替换算法主要在于两个函数，对于fifo，这两个函数为`_fifo_map_swappable`和`_fifo_swap_out_victim`

`_fifo_map_swappable`的逻辑很简单，维护一个双向链表，每次进来的页都从头指针的前一个或后一个位置进，这里采用加在头的后面

`_fifo_swap_out_victim`则是要找出要被换出的页，根据之前维护的双向链表，弹出头的前一个元素即可，然后将其从链表删除

(11)

对于 extended clock算法，引用`mmu.h`中的PTE_D、PTE_A分别表示修改位、访问位

`_clock_map_swappable`的逻辑很简单，维护一个双向链表，每次进来的页都从头指针的前一个或后一个位置进，这里采用加在头的后面,将对应表项的修改位、访问位清零

`_clock_swap_out_victim`则是要找出要被换出的页，根据之前维护的双向链表,进入循环根据算法规则，改变修改位、访问位，若找到二者皆为0的页，将其换出

(12)

pmm.c完成了物理内存的管理，在虚拟内存中，虚拟内存单元不一定有实际的物理内存单元对应，这样自然的需要一种数据结构去管理虚拟内存页，包括没有物理内存单元对应的合法虚拟页，当虚拟内存页被访问时再分配物理页帧，即按需分配