Lab11 虚拟内存管理、缺页中断及页面置换

一、 实验概述

在本次实验中，我们将实现虚拟内存管理，并处理缺页异常（Page Fault），实现页面置换算法。

二、实验目的

1. 了解虚拟内存的管理方式
2. 了解缺页中断
3. 实现页面置换

三、实验项目整体框架概述

```
// Lab11
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── consh
│   │   ├── ide.c//模拟硬盘
│   │   ├── ide.h
│   │   ├── intr.c
│   │   └── intr.h
│   ├── fs //模拟硬盘
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   │   ├── default_pmm.c
│   │   ├── default_pmm.h
│   │   ├── memlayout.h
│   │   ├── mmu.h
│   │   ├── pmm.c //更新了分配函数
│   │   ├── pmm.h
│   │   ├── swap.c//替换的相关实现
│   │   ├── swap_fifo.c //fifo 替换算法
│   │   ├── swap_fifo.h
│   │   ├── swap.h
│   │   ├── vmm.c  //虚拟内存管理相关信息
│   │   └── vmm.h
│   ├── sync
```

```
│   └── trap
│       ├── trap.c
│       ├── trap.h
│       └── trapentry.S
├── libs
├── Makefile
└── tools
```

四、实验内容

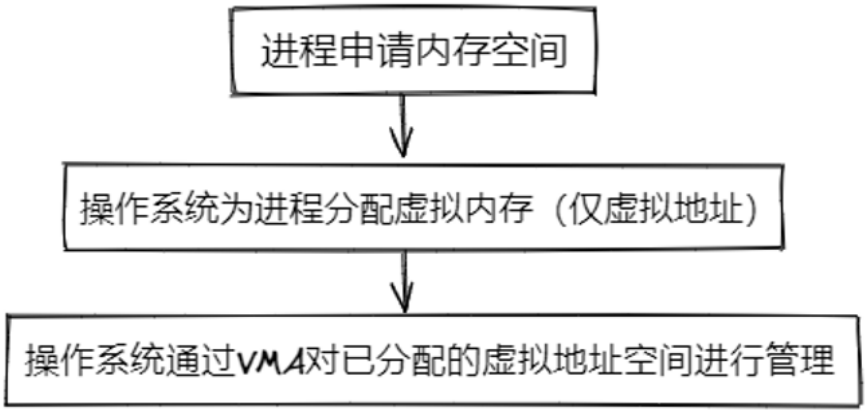
1. 理解VMA及其代码实现
2. 理解page fault和相关处理代码
3. 学习页面置换算法并进行实现

五、实验过程概述及相关知识点

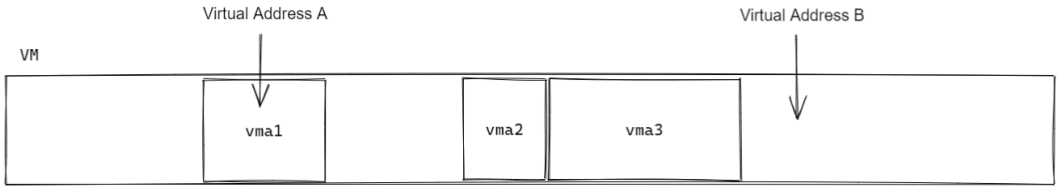
本次实验我们将了解虚拟内存（地址）的管理机制，以及触发缺页中断时操作系统应该如何处理缺页中断。

首先我们需要了解进程内存管理的大致处理过程：

1. 进程申请内存空间



> 在进程动态申请内存时，操作系统会首先为进程分配一块线性地址的使用权，而非一块实际的物理内存，只有进程真正需要操作该地址对应的内存时，才会通过触发缺页异常从而分配实际的物理内存并建立物理地址和虚拟地址的映射。
>
> 这个使用权如何分配呢？用户进程的虚拟空间实际是分为若干连续地址块的，每块空间的使用权限根据其内容而不同，操作系统为每个不同的空间块设计了一个VMA结构体用于管理对应的空间地址。每个用户进程会有一个链表用于管理它自己的所有VMA结构体，进程向操作系统申请内存时，操作系统将找到可以分配的虚拟地址空间块并生成该空间块对应的VMA结构体，将该结构体插入用户进程的链表后，即代表该进程拥有该地址段的使用权。
>
> ### Virtual memory areas
>
> 一个VMA结构体是一块连续的虚拟地址空间（Virtual memory areas）的抽象，它包含地址的描述（起始地址）以及该地址空间的权限描述等。

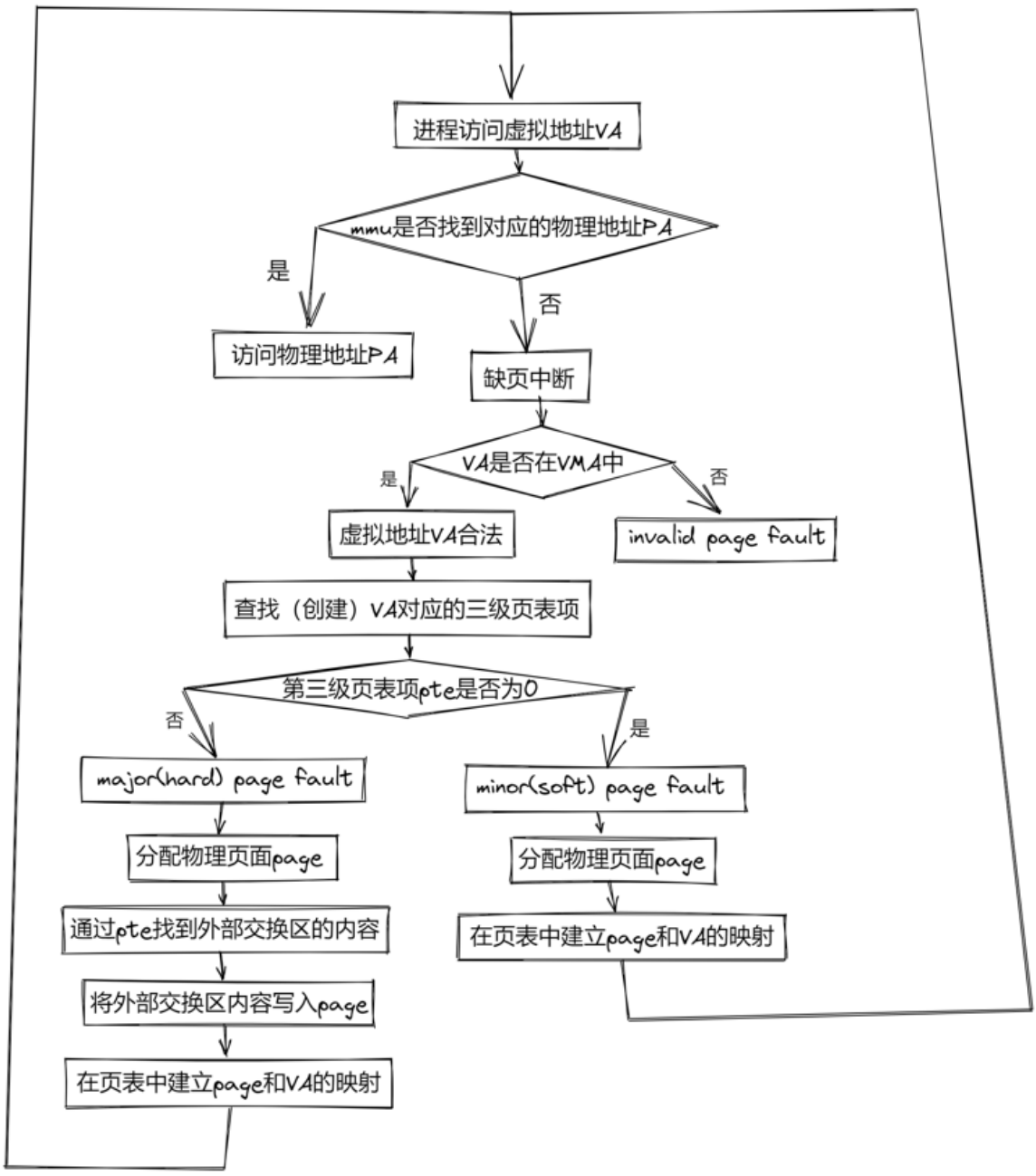


如图所示，每个进程都会有自己的虚拟地址空间布局，其中属于VMA的地址段才是这个进程可用的虚拟地址（享有使用权），图中Virtual address A是一个合法的虚拟地址，而Virtual address B则是不可用的。

本次实验中，我们会实现对VMA进行管理的代码。

2. 进程访问虚拟内存空间

当进程需要访问一个虚拟地址时，处理的流程如下图所示：



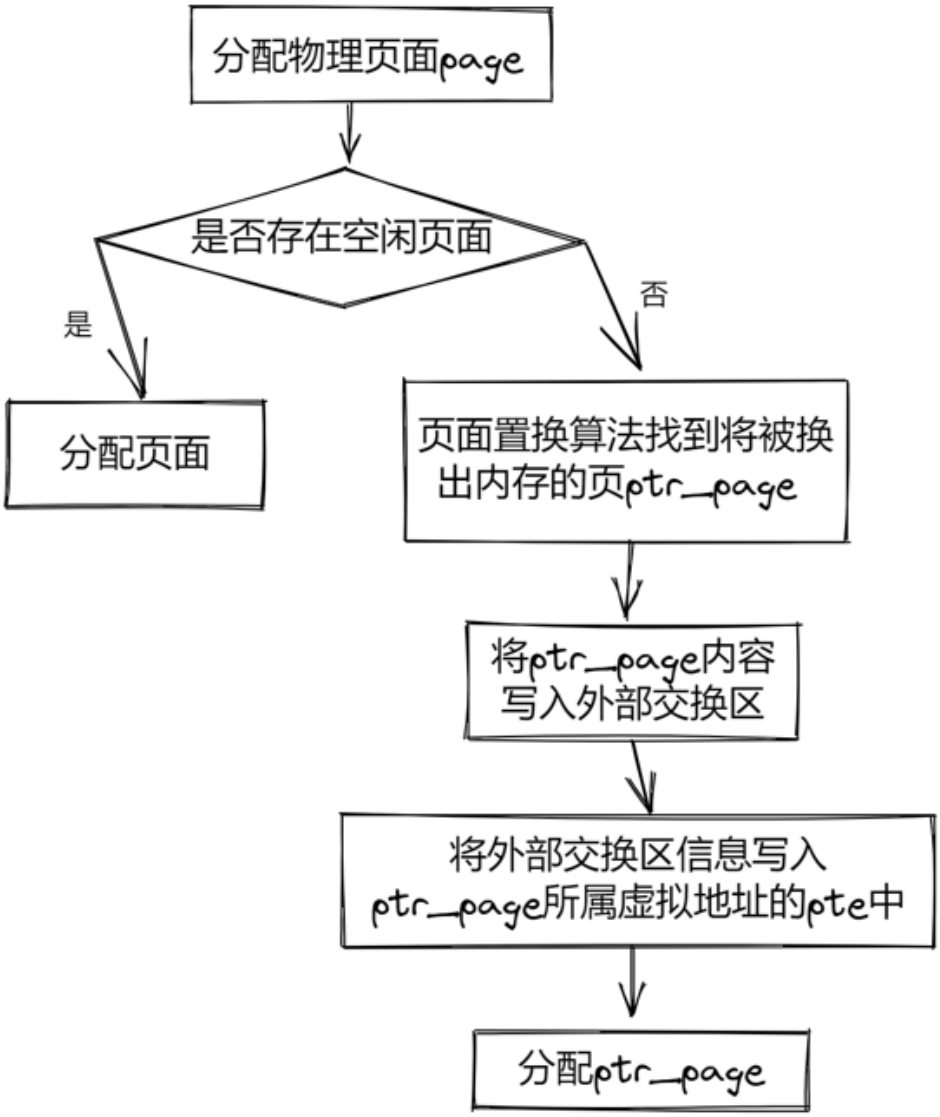
> 前面我们提到当进程申请内存时，操作系统会为进程提供一段虚拟地址的使用权，但并没有实际分配虚拟地址对应的物理内存。而当进程访问该地址时，会发现虚拟地址并没有对应的物理地址，从而触发缺页中断（page fault），而后操作系统通过响应该中断实现物理内存的分配，并在页表中建立相应的地址映射。
>
> ### page fault
>
> 当cpu访问虚拟地址，而该虚拟地址找不到对应的物理内存时触发该异常。以下情况可能导致page fault被触发：
>
> 1. 页表中没有虚拟地址对应的PTE（虚拟地址无效或虚拟地址有效但没有分配物理内存页）
> 2. 现有权限无法操作对应的PTE
>
> linux中缺页中断分为三种类型：
>
> 1. major page fault (hard page fault)
>
> 访问的虚拟地址内容不在内存中，需要从外设载入。常见于内容页被置换到外设交换区中，需要将交换区中的页面重新载入内存。
>
> 2. minor page fault (soft page fault)
>
> 虚拟地址在页表中没有建立映射，常见于进程申请虚拟内存后初次操作内存，及多个进程访问共享内存尚未建立虚拟地址映射的情况。
>
> 3. invalid fault
>
> 访问的虚拟地址不合法。

本次实验中，我们会实现对page fault进行响应及处理的代码。

注：由于没有硬盘交换区，实验中会使用物理内存模拟出一块硬盘，模拟硬盘不是本次实验的重点。

3. 分配页面，页面置换

当需要在物理内存分配页面时，我们需要调用之前物理内存管理实验中分配页面的方法。但是在之前的实验代码中，我们认为是存在空闲页面的，因而未处理过没有空闲页面的情况。当没有空闲页面，又需要分配新的页面时，则会需要进行页面置换，即从现有页面中选出一页，将其内容放入交换区，再把该页面分配给进程。具体流程如下：



页面置换算法

从当前页面选择被置换出的具体页面时，我们希望把即将使用的页面（常用）尽量保留，即尽量选择最近不会被使用的页面（不常用），从而减少page fault的产生，因而产生了各种算法用于选择页面：

- 先进先出(First In First Out, FIFO)页替换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象（Belady 现象），即在增加放置页的物理页帧的情况下，反而使页访问异常次数增多。
- 最久未使用(least recently used, LRU)算法：利用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。
- 时钟（Clock）页替换算法：是 LRU 算法的一种近似实现。时钟页替换算法把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针（简称当前指针）指向最老的那个页面，即最先进来的那个页面。另外，时钟算法需要在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU 中的 MMU 硬件将把

> 访问位置“1”。当操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为“0”，则淘汰该页，如果该页被写过，则还要把它换出到硬盘上；如果访问位为“1”，则将该页表项的此位置“0”，继续访问下一个页。该算法近似地体现了 LRU 的思想，且易于实现，开销少，需要硬件支持来设置访问位。时钟页替换算法在本质上与 FIFO 算法是类似的，不同之处是在时钟页替换算法中跳过了访问位为 1 的页。
> - 改进的时钟（Enhanced Clock）页替换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面，所以优先淘汰没有修改的页，减少磁盘操作次数。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中的 MMU 硬件将把访问位置“1”。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。

本次实验代码会实现页面置换算法中的fifo算法。

六、代码实现

1.VMA虚拟内存管理

mm和vma结构体

每个进程都有自己的虚拟地址空间，在前面的实验中，我们实现里物理内存的管理和页表。对于虚拟内存，我们定义两个结构体 mm_struct 和 vma_struct来管理虚拟地址空间。

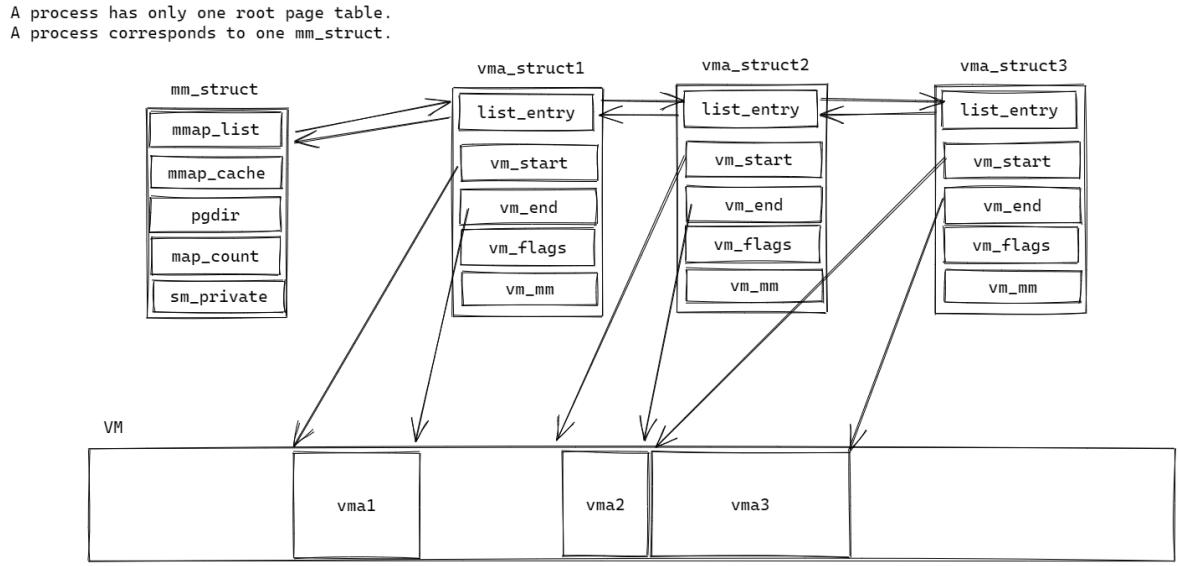
```
// the virtual continuous memory area(vma), [vm_start, vm_end),
// addr belong to a vma means  vma.vm_start<= addr <vma.vm_end
struct vma_struct {
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    uintptr_t vm_start;      // start addr of vma
    uintptr_t vm_end;        // end addr of vma, not include the vm_end itself
    uint_t vm_flags;       // flags of vma
    list_entry_t list_link;  // linear list link which sorted by start addr of
vma
};


// the control struct for a set of vma using the same Page Table
struct mm_struct {
    list_entry_t mmap_list;        // linear list link which sorted by start
addr of vma
    struct vma_struct *mmap_cache; // current accessed vma, used for speed
purpose
    pde_t *pgdir;                  // the Page Table of these vma
    int map_count;                 // the count of these vma
```

`vma_struct`结构体描述一段连续的虚拟地址，从`vm_start`到`vm_end`。 通过包含一个`list_entry_t`成员，我们可以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。`vm_flags`表示的是一段虚拟地址对应的权限（可读，可写，可执行等），这个权限在页表项里也要进行对应的设置。

我们注意到，每个页表（每个虚拟地址空间）可能包含多个`vma_struct`，也就是多个访问权限可能不同的，不相交的连续地址区间。我们用`mm_struct`结构体把一个页表对应的信息组合起来，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。

mm_struct和vma_struct的关系如下图所示：



mm和vma结构体对应的方法

创建两个结构体

```
// kern/mm/vmm.c
// mm_create -  alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    //在物理内存中分配一个mm_struct
    struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));

    if (mm != NULL) {
        list_init(&(mm->mmap_list));
        mm->mmap_cache = NULL;
        mm->pgdir = NULL;
        mm->map_count = 0;

        //这里为我们的页面置换算法所准备
        if (swap_init_ok) swap_init_mm(mm);
        else mm->sm_priv = NULL;
    }
    return mm;
}

// vma_create - alloc a vma_struct & initialize it. (addr range:
vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint_t vm_flags) {
    struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));
    if (vma != NULL) {
```

```
        vma->vm_start = vm_start;
        vma->vm_end = vm_end;
        vma->vm_flags = vm_flags;
    }
    return vma;
}
```

检查重叠

这里函数用来保证我们插入的vma对应的地址空间跟其他的vma对应的地址空间不会产生重叠

```
// kern/mm/vmm.c
// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end);// next 是我们想插入的区间，这里顺便检验了
start < end
}
```

将一个vma插入到mm_struct指向的链表中

mm_struct指向的vma结构体是按地址由小到大排列的，所以插入时首先根据vma中的start找到对应位置，然后检查是否跟前后的vma产生重叠，最后进行插入操作。

```
// insert_vma_struct -insert vma in mm's list link
void
insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
    assert(vma->vm_start < vma->vm_end);
    list_entry_t *list = &(mm->mmap_list);
    list_entry_t *le_prev = list, *le_next;

        list_entry_t *le = list;
        while ((le = list_next(le)) != list) {
            struct vma_struct *mmap_prev = le2vma(le, list_link);
            if (mmap_prev->vm_start > vma->vm_start) {
                break;
            }
            le_prev = le;
        }
    //保证插入后所有vma_struct按照区间左端点有序排列
    le_next = list_next(le_prev);

    /* check overlap */
    if (le_prev != list) {
        check_vma_overlap(le2vma(le_prev, list_link), vma);
    }
    if (le_next != list) {
        check_vma_overlap(vma, le2vma(le_next, list_link));
    }

    vma->vm_mm = mm;
    list_add_after(le_prev, &(vma->list_link));

    mm->map_count ++;//计数器
}
```

查找某个虚拟地址

这个函数的作用是查找某个虚拟地址是否存在在某个vma的start和end里，若存在返回对应的vma。

```
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;
}
```

2.处理缺页中断

> **模拟硬盘**
>
> 我们在QEMU里实际上并没有真正模拟“硬盘”。为了实现“页面置换”的效果，我们采取的措施是，从内核的静态存储(static)区里面分出一块内存， 声称这块存储区域是”硬盘“，然后包裹一下给出”硬盘IO“的接口。思考一下，内存和硬盘，除了一个掉电后数据易失一个不易失，一个访问快一个访问慢，其实并没有本质的区别。对于我们的页面置换算法来说，也不要求硬盘上存多余页面的交换空间能够“不易失”，反正这些页面存在内存里的时候就是易失的。那么我们就把QEMU模拟出来的一块ram叫做“硬盘”，用作页面置换时的交换区，完全没有问题。你可能会觉得，这样做，我们总共能使用的页面数并没有增加，原先能直接在内存里使用的一些页面变成了“硬盘”，只是在自娱自乐。确实，我们在这里只是想介绍页面置换的原理，并不关心实际性能。
>
> 这一部分我们在`driver/ide.h` `driver/ide.c` `fs/fs.h` `fs/swapfs.h` `fs/swapfs.c`实现。
>
> `fs`就是file system,我们这里其实并没有“文件”的概念，这个模块称作`fs`只是说明它是“硬盘”和内核之间的接口。
>
> `ide`在这里不是integrated development environment的意思，而是Integrated Drive Electronics的意思，表示的是一种标准的硬盘接口。我们这里写的东西和Integrated Drive Electronics并不相关，这个命名是ucore的历史遗留。
>
> 具体的“硬盘IO”代码就是基本的内存复制，我们不做详细的介绍。同时为了逼真地模仿磁盘，我们只允许以磁盘扇区为数据传输的基本单位，也就是一次传输的数据必须是512字节的倍数，并且必须对齐。

当我们引入了虚拟内存，就意味着虚拟内存的空间可以远远大于物理内存，意味着程序可以访问"不对应物理内存页帧的虚拟内存地址"，这时CPU应当抛出`Page Fault`这个异常。

回想一下，我们处理异常的时候，是在`kern/trap/trap.c`的`exception_handler()`函数里进行的。按照`scause`寄存器对异常的分类里，有`CAUSE_LOAD_PAGE_FAULT`和`CAUSE_STORE_PAGE_FAULT`两个case。之前我们并没有真正对异常进行处理，只是简单输出一下就返回了。现在我们要真正进行Page Fault的处理。

当CPU检测到Page Fault的时候，会把异常的的原因放在scause寄存器中，把对应的虚拟地址放在badvaddr（另一个名字叫stval）寄存器中。

```
// kern/trap/trap.c

void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        /* .... other cases */
        case CAUSE_FETCH_PAGE_FAULT:
            cprintf("Instruction page fault\n");
            break;
        case CAUSE_LOAD_PAGE_FAULT:
            cprintf("Load page fault\n");
            if ((ret = pgfault_handler(tf)) != 0) {
                print_trapframe(tf);
                panic("handle pgfault failed. %e\n", ret);
            }
            break;
        case CAUSE_STORE_PAGE_FAULT:
            cprintf("Store/AMO page fault\n");
            if ((ret = pgfault_handler(tf)) != 0) { //do_pgfault()页面置换成功时返回0
                print_trapframe(tf);
                panic("handle pgfault failed. %e\n", ret);
            }
            break;
        default:
            print_trapframe(tf);
            break;
    }
}

static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}
```

最终，异常会交给do_pgfault这个函数进行处理。

我们首先要做的就是在mm_struct里判断产生page fault的虚拟地址是否是有效的，如果有效，将这个虚拟地址按照页面的大小进行对齐，然后通过get_pte函数获取到对应的三级页表项，如果页表项为0，则分配一个新的物理页面（分配时如果物理内存满了，则会进行换出操作，这里我们修改了pmm.c中的alloc_pages这个函数），并建立相关的映射。如果页表项不为0，则说明对应的页面被换出了，我们需要根据pte在硬盘上找到对应的页面，并页面换入，建立映射，最后标记这个页面时可以换出的。

```
// kern/mm/vmm.c
int do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    //addr: 访问出错的虚拟地址
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);
    pgfault_num++;
    //If the addr is not in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }

    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE); //按照页面大小把地址对齐

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;

    ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                         //PT(Page Table) isn't existed, then
                                         //create a PT.
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {
        if (swap_init_ok) {
            struct Page *page = NULL;
            //在swap_in()函数执行完之后，page保存换入的物理页面。
            //swap_in()函数里面可能把内存里原有的页面换出去
            swap_in(mm, addr, &page);  //(1）According to the mm AND addr, try
                                       //to load the content of right disk page
                                       //into the memory which page managed.
            page_insert(mm->pgdir, page, addr, perm); //更新页表，插入新的页表项
            //(2) According to the mm, addr AND page,
            // setup the map of phy addr <---> virtual addr
            swap_map_swappable(mm, addr, page, 1);  //(3) make the page
swappable.
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
   }

   ret = 0;
failed:
    return ret;
}
```

3.页面置换

换入换出的具体实现

换入的时候，我们新分配一个物理页面，根据虚拟地址对应的pte在硬盘上找到对应的位置（这里用的时简单的偏移实现），将硬盘的内容换入到新分配的页面。

```
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();//这里alloc_page()内部可能调用swap_out()
     //找到对应的一个物理页面
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);//找到/构建对应的页表项
    //将物理地址映射到虚拟地址是在swap_in()退出之后，调用page_insert()完成的
     int r;
     if ((r = swapfs_read((*ptep), result)) != 0)//将数据从硬盘读到内存
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n",
(*ptep)>>8, addr);
     *ptr_result=result;
     return 0;
}
```

换出的时候，首先根据替换算法找到要被换出的那一个page，称为victim。接着找到victim对应的虚拟地址，根据这个虚拟地址构造一个可以放在硬盘上的位置，将victim换出。同时在更改victim对应的页表项，将硬盘对应的位置放入到victim的页表项中，最后释放掉对应的物理页面。

```
int swap_out(struct mm_struct *mm, int n, int in_tick)
{
     int i;
     for (i = 0; i != n; ++ i)
     {
          uintptr_t v;
          struct Page *page;
          int r = sm->swap_out_victim(mm, &page, in_tick);//调用页面置换算法的接口
          //r=0表示成功找到了可以换出去的页面
         //要换出去的物理页面存在page里
          if (r != 0) {
                  cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                  break;
          }

          cprintf("SWAP: choose victim page 0x%08x\n", page);

          v=page->pra_vaddr;//可以获取物理页面对应的虚拟地址
          pte_t *ptep = get_pte(mm->pgdir, v, 0);
          assert((*ptep & PTE_V) != 0);

          if (swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page) != 0) {
                      //尝试把要换出的物理页面写到硬盘上的交换区，返回值不为0说明失败了
                  cprintf("SWAP: failed to save\n");
                  sm->map_swappable(mm, v, page, 0);
                  continue;
```

```
            }
            else {
                //成功换出
                    cprintf("swap_out: i %d, store page in vaddr 0x%x to disk
swap entry %d\n", i, v, page->pra_vaddr/PGSIZE+1);
                    *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
                    free_page(page);
            }
            //由于页表改变了，需要刷新TLB
            tlb_invalidate(mm->pgdir, v);
        }
        return i;
}
```

替换管理器

类似 `pmm_manager`，我们定义 `swap_manager`，组合页面置换需要的一些函数接口。

```
struct swap_manager
{
     const char *name;
     /* Global initialization for the swap manager */
     int (*init)            (void);
     /* Initialize the priv data inside mm_struct */
     int (*init_mm)         (struct mm_struct *mm);
     /* Called when tick interrupt occured */
     int (*tick_event)      (struct mm_struct *mm);
     /* Called when map a swappable page into the mm_struct */
     int (*map_swappable)   (struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in);
     /* When a page is marked as shared, this routine is called to
      * delete the addr entry from the swap manager */
     int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
     /* Try to swap out a page, return then victim */
     int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page, int
in_tick);
     /* check the page relpacement algorithm */
     int (*check_swap)(void);
};
```

在我们完成替换功能的初始化的时候，我们首先初始化我们的硬盘，设置对应替换算法的管理器，然后把替换的全局标志位swap_init_ok设置为1，这样标志着替换功能已经初始化完成。

```
// kern/mm/swap.c
static struct swap_manager *sm;
int swap_init(void)
{
     swapfs_init();

     // Since the IDE is faked, it can only store 7 pages at most to pass the
test
     if (!(7 <= max_swap_offset &&
        max_swap_offset < MAX_SWAP_OFFSET_LIMIT)) {
        panic("bad max_swap_offset %08x.\n", max_swap_offset);
     }
```

```
     sm = &swap_manager_fifo;//use first in first out Page Replacement Algorithm
     int r = sm->init();

     if (r == 0)
     {
          swap_init_ok = 1;
          cprintf("SWAP: manager = %s\n", sm->name);
          check_swap();
     }

     return r;
}

int swap_init_mm(struct mm_struct *mm)
{
     return sm->init_mm(mm);
}

int swap_tick_event(struct mm_struct *mm)
{
     return sm->tick_event(mm);
}

int swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
int swap_in)
{
     return sm->map_swappable(mm, addr, page, swap_in);
}

int swap_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
     return sm->set_unswappable(mm, addr);
}
```

替换算法的实现

这里我们以FIFO算法进行举例

FIFO(First in, First out)页面置换算法，就是把所有页面排在一个队列里，每次换入页面的时候，把队列里最靠前（最早被换入）的页面置换出去。`kern/mm/swap_fifo.c`完成了FIFO置换算法最终的具体实现。我们所做的就是维护了一个队列（用链表实现）。

_fifo_init_mm：初始化一个FIFO队列，并且将mm_struct中的sm_priv指向这个队列。

_fifo_map_swappable：将对应的page设置成为一个可以被替换的页面，即将page->pra_page_link的插入到FIFO队列。

_fifo_swap_out_victim：根据FIFO算法，选取要被换出的物理页面。

```
// kern/mm/swap_fifo.h
#ifndef __KERN_MM_SWAP_FIFO_H__
#define __KERN_MM_SWAP_FIFO_H__

#include <swap.h>
extern struct swap_manager swap_manager_fifo;

#endif
// kern/mm/swap_fifo.c
```

```
 /* In order to implement FIFO PRA, we should manage all swappable pages, so we
can link these pages into pra_list_head according the time order.
 */

list_entry_t pra_list_head;
/*
 *  _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr of
pra_list_head.
 *     Now, From the memory control struct mm_struct, we can access FIFO PRA
 */
static int
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     return 0;
}
/*
 * _fifo_map_swappable: According FIFO PRA, we should link the most recent
arrival page at the back of pra_list_head qeueue
 */
static int
_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && head != NULL);
    //record the page access situlation
    //link the most recent arrival page at the back of the pra_list_head qeueue.
    list_add(head, entry);
    return 0;
}
/*
 *  _fifo_swap_out_victim: According FIFO PRA, we should unlink the  earliest
arrival page in front of pra_list_head qeueue,
 *    then set the addr of addr of this page to ptr_page.
 */
static int
_fifo_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);
     /* Select the victim */
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  set the addr of addr of this page to ptr_page
    list_entry_t* entry = list_prev(head);
    if (entry != head) {
        list_del(entry);
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        *ptr_page = NULL;
    }
    return 0;
}
```

```
struct swap_manager swap_manager_fifo =
{
     .name            = "fifo swap manager",
     .init            = &_fifo_init,
     .init_mm         = &_fifo_init_mm,
     .tick_event      = &_fifo_tick_event,
     .map_swappable   = &_fifo_map_swappable,
     .set_unswappable = &_fifo_set_unswappable,
     .swap_out_victim = &_fifo_swap_out_victim,
     .check_swap      = &_fifo_check_swap,
};
```

4.测试函数简介

`check_vma_struct`：主要检测我们实现的两个结构体以及对应的方法是否满足要求。

涉及的具体函数：mm_create()，vma_create()， insert_vma_struct()，find_vma()

`check_pgfault`：在这个测试函数中，会使一个有效的虚拟地址虚拟地址产生page fault，检查page fault的处理。

`check_swap`和`_fifo_check_swap`：在这两个测试函数中，我们首先分配四个物理页面，然后把free_list清空，即我们只有四个物理页面。有效的虚拟地址为[0x1000,0x6000)，对应于5个虚拟页面，这样就会产生硬缺页中断。我们按照一定的顺序分别使用五个虚拟页面的地址，检查FIFO算法的正确性。

七、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 虚拟内存管理方式
2. PageFault的产生原因及处理过程
3. 简单页面置换算法的实现

八、下一实验简单介绍

下一步，我们将进入进程管理的部分。