Lab11 页表映射、虚拟内存管理、缺页中断

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）。

二、实验目的

1. 掌握页表的建立和使用方法
2. 了解虚拟内存的管理方式
3. 了解缺页中断

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

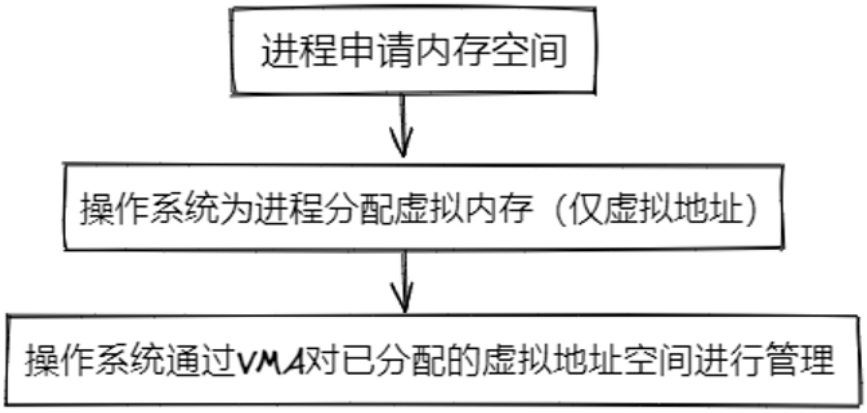
1. 掌握用户空间如何根据虚拟地址与物理地址创建映射（页表）
2. 掌握如何删除页表
3. 理解VMA及其代码实现
4. 理解page fault和相关处理代码

五、实验过程概述及相关知识点

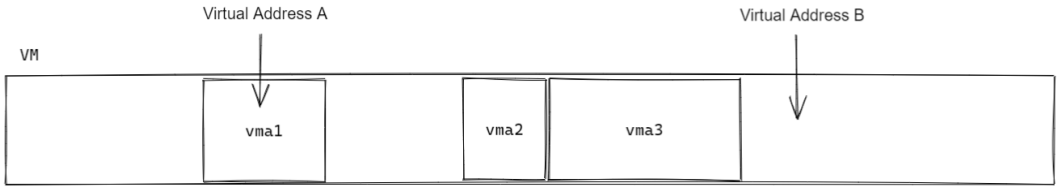
本次实验我们将了解虚拟内存（地址）的管理机制，以及触发缺页中断时操作系统应该如何处理缺页中断。

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。

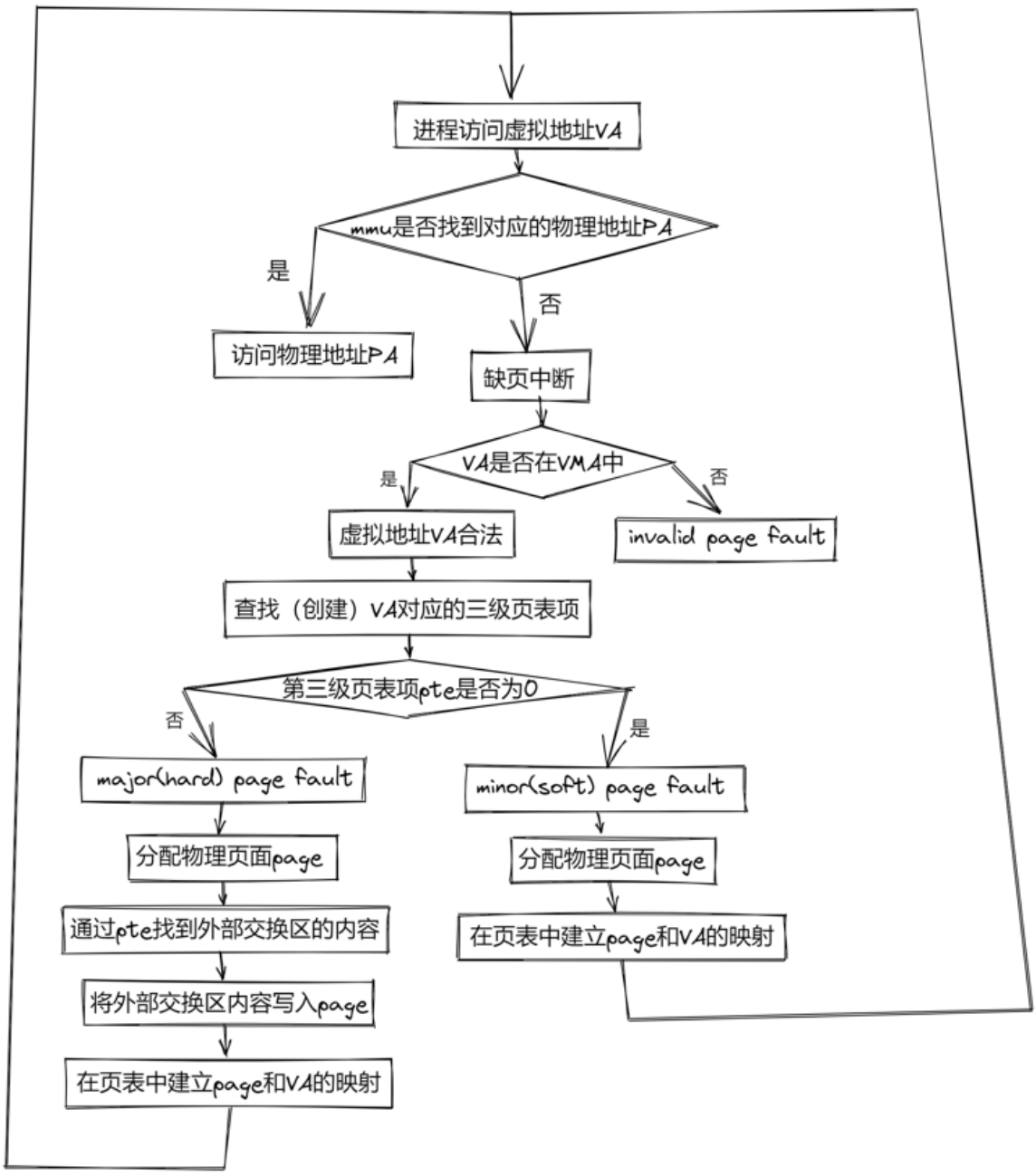


如图所示，每个进程都会有自己的虚拟地址空间布局，其中属于VMA的地址段才是这个进程可用的虚拟地址（享有使用权），图中Virtual address A是一个合法的虚拟地址，而Virtual address B则是不可用的。

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。

六、代码实现

Step1. 建立多级页表，填写页表项

我们需要把页表放在内存里，并且需要有办法修改页表，比如在页表里增加一个页面的映射或者删除某个页面的映射。

当得到一个虚拟地址后，我们会为其分配对应的物理页面，内核可以通过吉页的地址映射方式简单的得到页面对应的物理地址，我们要做的是把虚拟地址到物理地址的三级页表映射写入实际的页表中。

这部分操作最主要涉及的是两个接口：

`page_insert()`，在页表里建立一个映射

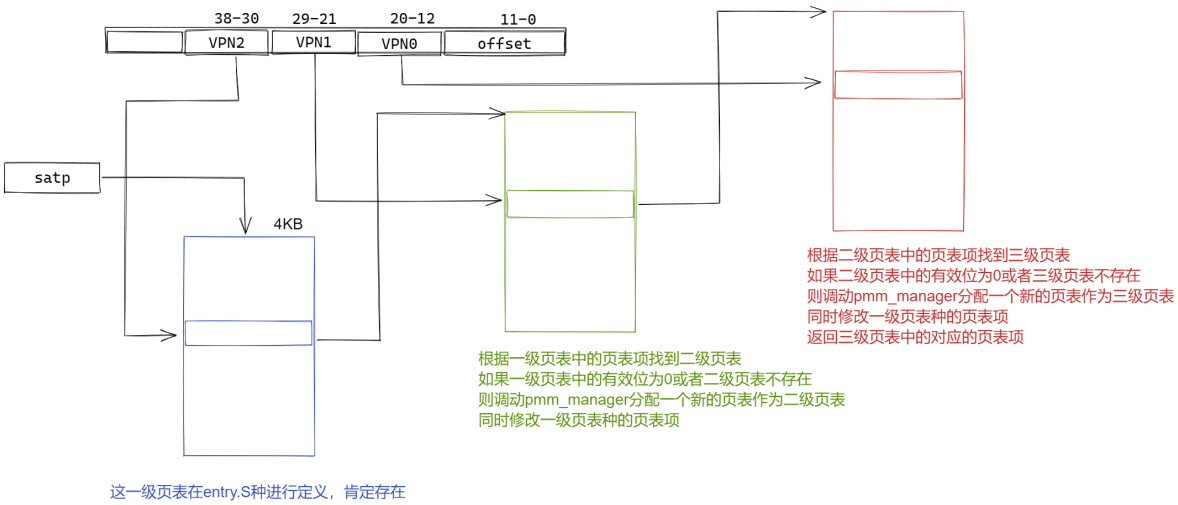
`page_remove()`，在页表里删除一个映射

这些我们都在`kern/mm/pmm.c`里面编写。

我们来看`page_insert()`,`page_remove()`的实现。注意它们都要调用两个对页表项进行操作的函数：`get_pte()`和`page_remove_pte()`

1.get_pte()

get_pte 根据虚拟地址(在代码中写为la，线性地址)找到（创建）对应的三级页表项




```
//寻找(有必要的时候分配)一个页表项
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /*
     * If you need to visit a physical address, please use KADDR()
     * please read pmm.h for useful macros
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
     *   KADDR(pa) : takes a physical address and returns the corresponding
     * kernel virtual address.
     *   set_page_ref(page,1) : means the page be referenced by one time
     *   page2pa(page): get the physical address of memory which this (struct
     * Page *) page  manages
     *   struct Page * alloc_page() : allocation a page
     *   memset(void *s, char c, size_t n) : sets the first n bytes of the
     * memory area pointed by s
     *                                       to the specified value c.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
     * flags bit : Present
     *   PTE_W           0x002                   // page table/directory entry
     * flags bit : Writeable
     *   PTE_U           0x004                   // page table/directory entry
     * flags bit : User can access
     */
    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) {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        //我们现在在虚拟地址空间中，所以要转化为KADDR再memset.
        //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
成对物理内存的访问。
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//注意这里R,W,X全零
    }
```

```
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//再下一级页表
    //这里的逻辑和前面完全一致，页表不存在就现在分配一个
    if (!(*pdep0 & PTE_V)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
                return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    //找到输入的虚拟地址la对应的页表项的地址(可能是刚刚分配的)
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

2. page_remove_pte()

这个函数是给定根页表，对应的页表项和线性地址，删除这个页表项以及它映射的page。

首先会判断页表项里的有效位，如果为0，则不执行任何操作。如果有效，则进行删除操作。获取到对应的Page结构体，ref值减1，如果ref值为0了，则释放掉这个page结构体，页表项归0，刷新快表。

```
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_V) {  //(1) check if this page table entry is valid
        struct Page *page = pte2page(*ptep);  //(2) find corresponding page to
pte
        page_ref_dec(page);   //(3) decrease page reference
        if (page_ref(page) == 0) {
            //(4) and free this page when page reference reachs 0
            free_page(page);
        }
        *ptep = 0;                  //(5) clear page table entry
        tlb_invalidate(pgdir, la);  //(6) flush tlb
    }
}
```

3.根据虚拟地址与物理地址创建映射（page_insert)

```
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    //pgdir是页表基址(satp)，page是分配的物理页面的对应结构体的虚拟地址，la是虚拟地址
    pte_t *ptep = get_pte(pgdir, la, 1);
    //先找到对应页表项的位置，如果原先不存在，get_pte()会分配页表项的内存
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page);//指向这个物理页面的虚拟地址增加了一个
    if (*ptep & PTE_V) { //原先存在映射
        struct Page *p = pte2page(*ptep);
        if (p == page) {//如果这个映射原先就有
            page_ref_dec(page);
        } else {//如果原先这个虚拟地址映射到其他物理页面，那么需要删除映射
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = pte_create(page2ppn(page), PTE_V | perm);//构造页表项
    tlb_invalidate(pgdir, la);//页表改变之后要刷新TLB
```

```
    return 0;
}
```

这个函数的作用是给定一个虚拟地址，根页表（satp寄存器指向的页表），一个物理内存分配器分配出来的物理页面及对应的标志位，将页面与虚拟地址对应起来。如果发现虚拟地址对应于其他的页面，则将第三级页表中的页表项进行替换，最后刷新块表。

第一步：get_pte 根据虚拟地址(在代码种写为la，线性地址)找到对应的三级页表项

第二步：判断获得的页表项是否对应于给定的页面，如果不对应，则删除原来的页表项以及释放掉对应的page的结构体（相当于页面被free掉）。

第三步：设置对应的页表项，并且刷新快表。

4.删除映射（page_remove)

```
void page_remove(pde_t *pgdir, uintptr_t la) {
    pte_t *ptep = get_pte(pgdir, la, 0);//找到页表项所在位置
    if (ptep != NULL) {
        page_remove_pte(pgdir, la, ptep);//删除这个页表项的映射
    }
}
```

Step2. 建立页表映射测试函数介绍

本次实验中主要有两个函数进行测试：`check_pgdir()`和`check_boot_pgdir()`

`check_pgdir()`主要对get_page，page_insert，get_pte，page_remove等函数进行测试。

在这其中用到了两个模拟的虚拟地址，一个是0x0，另一个是PGSIZE也就是4096这个地址，进行page_insert，然后判断pte以及标志位是否设置正确，page_ref是否正确，进行page_remove，重新判断page_ref。

`check_boot_pgdir()`主要进行的测试是：

有三个虚拟地址0x100，0x100+PGSIZE，page2kva(p) + 0x100，以及一个物理页面（p指针指向这个物理页面对应的struct page）。

之后通过page_insert()在sv39的三级页表中设置对应的页表项使得0x100，0x100+PGSIZE都指向p所指向的页面偏移0x100的地址（0x100小于4096，因此是页面的偏移量）。

而page2kva(p) + 0x100是通过我们设置的内核虚拟地址的大页映射方式取得的p对应的物理页面的虚拟地址，再加上偏移量也就是p锁指向的页面偏移0x100的地址。

因此三个虚拟地址指向了同一个物理空间。测试中改变了其中一个虚拟空间的内容，再去查看另外两个虚拟地址空间的内容是不是也有相同的变化。

Step3.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),
```

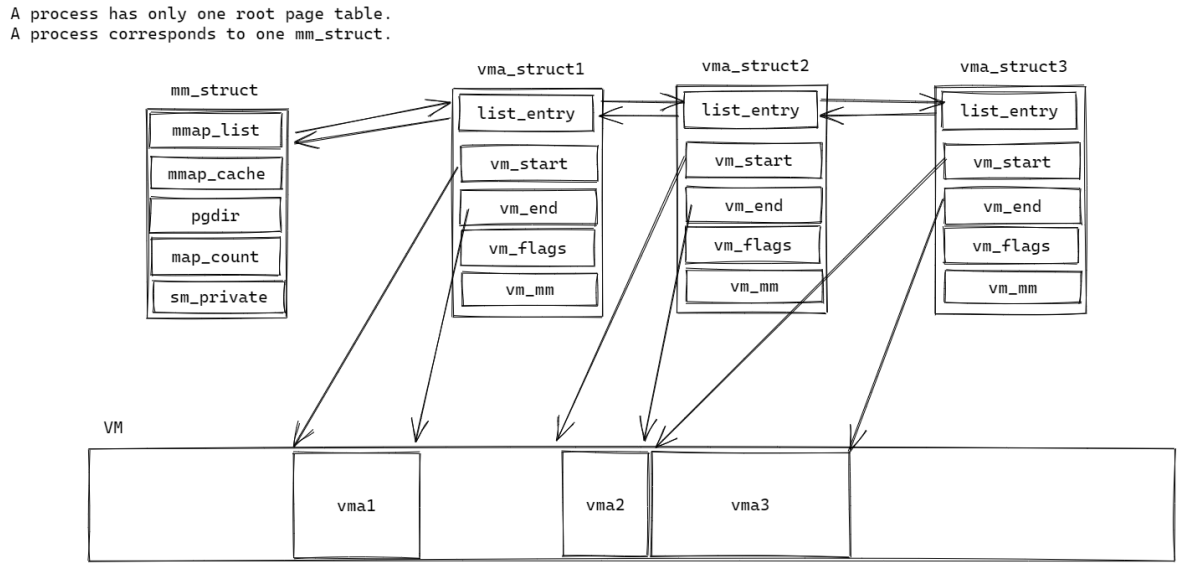
```
// addr belong to a vma means  vma.vm_start<= addr <vma.vm_end
struct vma_struct {
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    uintptr_t vm_start;      // start addr of vma
    uintptr_t vm_end;        // end addr of vma, not include the vm_end itself
    uint_t vm_flags;       // 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;
}
```

Step4.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;
}
```

Step5.缺页中断测试函数简介

`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。

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. 如何建立虚拟地址到物理地址的三级页表
2. 虚拟内存管理方式
3. PageFault的产生原因及处理过程

八、下一实验简单介绍

下一步，我们将学习页面置换的部分。