lab3:缺页异常和页面置换

实验目的

- 了解虚拟内存的Page Fault异常处理实现
- 了解页替换算法在操作系统中的实现
- 学会如何使用多级页表，处理缺页异常（Page Fault），实现页面置换算法。

实验内容

练习1：理解基于FIFO的页面替换算法（思考题）

> 描述FIFO页面置换算法下，一个页面从被换入到被换出的过程中，会经过代码里哪些函数/宏的处理（或者说，需要调用哪些函数/宏），并用简单的一两句话描述每个函数在过程中做了什么？（为了方便同学们完成练习，所以实际上我们的项目代码和实验指导的还是略有不同，例如我们将FIFO页面置换算法头文件的大部分代码放在了`kern/mm/swap_fifo.c`文件中，这点请同学们注意）
>
> - 至少正确指出10个不同的函数分别做了什么？如果少于10个将酌情给分。我们认为只要函数原型不同，就算两个不同的函数。要求指出对执行过程有实际影响,删去后会导致输出结果不同的函数（例如assert）而不是cprintf这样的函数。如果你选择的函数不能完整地体现”从换入到换出“的过程，比如10个函数都是页面换入的时候调用的，或者解释功能的时候只解释了这10个函数在页面换入时的功能，那么也会扣除一定的分数

1. `swap_init()`：**该函数进行初始化页面置换算法。**

   此处选定使用FIFO页面替换算法，并输出当前使用的页面置换算法为FIFO算法。（在页面置换前调用），然后调用`check_swap()`进行swap的检查。

```
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_clock;//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;
```

```
}
```

2. `exception_handler()`：**该函数检测缺页异常。（在页面置换前调用）**

当检测到当前异常为`CAUSE_LOAD_PAGE_FAULT`（页面读异常）或`CAUSE_STORE_PAGE_FAULT`（页面写异常）时，调用`pgfault_handler`进行缺页异常处理，`pgfault_handler`函数会调用`do_pgfault()`进行页面置换；如果`do_pgfault()`页面置换不成功则返回非0，此时输出"handle pgfault failed"。

```
void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        /*其他case省略...*/
        case CAUSE_FETCH_PAGE_FAULT: //取指令时发生的Page Fault先不处理
            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;
    }
```

3. `pgfault_handler()`：**该函数进行缺页异常处理。（在页面置换前调用）**

该函数首先打印了缺页异常发生的虚拟地址、是否发生在内核中、是否是`CAUSE_STORE_PAGE_FAULT`（页面写异常）；使用虚拟内存管理结构体`check_mm_struct`调用`do_pgfault()`进行页面置换。

```
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");
}
```

4. `do_pgfault()`：**该函数进行页面置换过程。（开始进行页面换入，同时可能有页面换出）**

该函数传入访问出错的虚拟地址`addr`，首先将`ret`初始化为无效参数`E_INVAL`（宏）；然后在`mm_struct`里使用`find_vma()`函数判断这个出错虚拟地址是否可用，缺页异常计数器`page fault number`加一；如果该出错虚拟地址无效或不可用，则输出提示信息，返回ret为`E_INVAL`；如果该出错虚拟地址可用，声明变量`perm`初始化权限为`PTE_U`(宏，user)和用户态的可读可写；接着按照页面大小把地址对齐，调用`get_pte()`寻找（不存在时分配）页表项，这里实际上页表项不存在，会新建一个页表项，然后进行页面置换：首先调用`swap_in()`分配一个内存页，然后根据PTE中的swap条目的`addr`，找到磁盘页的地址，将磁盘页的内容读入这个内存页，在执行完后，内存页保存换入的物理页面，且可能把内存原有的页面换出去；然后调用`page_insert()`更新页表，插入新的页表项，建立一个Page的phy addr与线性addr la的映射；最后调用`swap_map_swappable()`标记这个页面将来是可以再换出来的。

```
//进行页面置换
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    //addr为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);
    //首先在mm_struct里判断这个虚拟地址是否可用
    pgfault_num++;
    //If the addr is 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;
    }
    /* IF (write an existed addr ) OR
     *    (write an non_existed addr && addr is writable) OR
     *    (read  an non_existed addr && addr is readable)
     * THEN
     *    continue process
     */
    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 {
        /*LAB3 EXERCISE 3: YOUR CODE
        * 请你根据以下信息提示，补充函数
        * 现在我们认为pte是一个交换条目，那我们应该从磁盘加载数据并放到带有phy addr的页面，
        * 并将phy addr与逻辑addr映射，触发交换管理器记录该页面的访问情况
```

```
        *
        *  一些有用的宏和定义，可能会对你接下来代码的编写产生帮助(显然是有帮助的)宏或函数:
        *    swap_in(mm, addr, &page) : 分配一个内存页，然后根据PTE中的swap条目的addr，找到磁盘页的地址，将磁盘页的内容读入这个内存页
        *    page_insert : 建立一个Page的phy addr与线性addr la的映射
        *    swap_map_swappable : 设置页面可交换
        */
        if (swap_init_ok) {
            struct Page *page = NULL;
            // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
            //(1) According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            swap_in(mm,addr,&page); //在执完后，page保存换入的物理页面，可能把内存原有的页面换出去
            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 <---> logical addr
            swap_map_swappable(mm,addr,page,1); //(3) make the page swappable.
            //标记这个页面将来是可以再换出来的
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
   }

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

5. `find_vma()`：**在页面换入前对某个虚拟地址进行检查。**

    查找某个虚拟地址对应的vma_struct是否存在，如果返回null，说明查询的虚拟地址不存在/不合法，既不对应内存里的某个页，也不对应硬盘里某个可以换进来的页。

```
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;
}
```

5. `get_pte()`：**在页面换出、页面换入时均有调用，用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作，是实现虚拟内存与物理内存之间映射关系非常重要的内容。**

    在页面换入时调用该函数给指定的地址分配二级和一级页表项；在页面换出时调用该函数找到需要换出的页面对应的页表项。具体实现在第二问中会进行详细介绍。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep1 = &pgdir[PDX1(la)];
    if (!(*pdep1 & 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);
        //现在在虚拟地址空间中 ，要转化为KADDR再memset
        //不管页表怎么构造，我们要确保物理地址和虚拟地址的偏移量始终相同，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);
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
//    pde_t *pdep0 = &((pde_t *)(PDE_ADDR(*pdep1)))[PDX0(la)];
    //和前面逻辑一样，页表不存在就分配一个
    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);
 //      memset(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（可能是刚分配好的）
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

6. `swap_in()`：**实现换入一个页面。**

    首先调用`alloc_page()`（宏，对应`alloc_pages()`函数）进行页面分配（在`alloc_page()`内部可能调用`swap_out()`）；找到其对应的物理页面，然后找到构建对应的页表项，并调用`swapfs_read()`将物理地址的数据(视作硬盘)写入页表项(视作内存)，然后将获取的页返回。

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();  //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
     // cprintf("SWAP: load ptep %x swap entry %d to vaddr 0x%08x, page %x, No
%d\n", ptep, (*ptep)>>8, addr, result, (result-pages));

     //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)
     {
        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;
}
```

7. `alloc_pages()`：**为换入的页面进行物理页面分配。**

   如果有足够的物理页面，就不必换出其他页面；如果n>1，说明希望分配多个连续的页面，但换出页面的时候并不能换出连续的页面；如果页面置换算法没有成功初始化，则不能成功分配页面；如果没有足够的物理页面，则先调用`swap_out()`进行页面的换出。

```
struct Page *alloc_pages(size_t n) {
    struct Page *page = NULL;
    bool intr_flag;

    while (1) {
        local_intr_save(intr_flag);
        { page = pmm_manager->alloc_pages(n); }
        local_intr_restore(intr_flag);
        //如果有足够的物理页面，就不必换出其他页面
        //如果n>1，说明希望分配多个连续的页面，但换出页面的时候并不能换出连续的页面
        if (page != NULL || n > 1 || swap_init_ok == 0) break; //swap_init_ok标志
是否成功初始化

        extern struct mm_struct *check_mm_struct;
        // cprintf("page %x, call swap_out in alloc_pages %d\n",page, n);
        swap_out(check_mm_struct, n, 0);
    }
    // cprintf("n %d,get page %x, No %d in alloc_pages\n",n,page,(page-pages));
    return page;
}
```

7. `swap_out()`：**实现换出一个页面。**

首先调用了页面置换算法的接口，若`r=0`表示成功找到了可以换出去的页面，更换出去的物理页面存在page里，否则输出没有找到的提示信息，返回`i`。找到后获取物理页面对应的虚拟地址，并调用`swapfs_write()`尝试把要换出去的物理页面写到硬盘上的交换区，返回值为0说明成功了，输出换出信息，然后调用`free_page()`释放该页；否则失败，输出失败信息；最后由于页表改变了，需要刷新TLB。

```
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 **ptr_page=NULL;
          struct Page *page;
          // cprintf("i %d, SWAP: call swap_out_victim\n",i);
          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，更换除去的物理页面存在page里
          if (r != 0) {
                    cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                  break;
          }
          v=page->pra_vaddr; //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) {
                    //尝试把要换出去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;
}
```

8. `page_insert()`：**在页面换入之后，更新页表，插入新的页表项，建立物理地址与虚拟地址的映射。**

```
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    //pgdir是页表基址(satp)，page对应物理页面，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;
}
```

9. `swap_map_swappable()`：**在页面换入之后，标记这个页面将来是可以再换出来的。**

```
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);
}
```

10. `swapfs_read()`：**在页面换入时，调用**`ide_read_secs()`**函数实现物理地址的数据(视作硬盘)写入页表项(视作内存)**

```
int swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

11. `swapfs_write()`：**在页面换出时，调用**`ide_write_secs()`**函数实现将要换出去的物理页面写到硬盘上的交换区。**

```
int swapfs_write(swap_entry_t entry, struct Page *page) {
    return ide_write_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

12. `swap_out_victim()`：**在页面换出时，调用该函数，该函数在每个页面置换管理结构体**`swap_manager`**的成员函数中实现，找出应该换出的页面。**

```
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;
    }
```

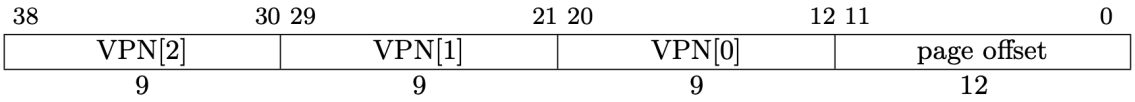
练习2：深入理解不同分页模式的工作原理（思考题）

> get_pte()函数（位于`kern/mm/pmm.c`）用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作。这在操作系统中的分页机制下，是实现虚拟内存与物理内存之间映射关系非常重要的内容。
>
> - 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。
> - 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？

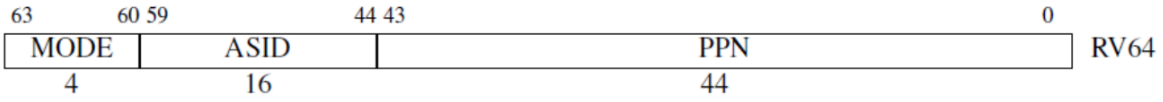
- 首先，我们先来理解一下 `sv32`，`sv39`，和 `sv48` 三种不同的页表级别的异同。

	sv32	**sv39**	**sv48**
页表项位数	32位	39位	48位
虚拟地址空间	4 GB	512 GB	256 TB
页表级别	32 位的虚拟地址空间通常被分为12位的页内偏移和两个10位的页号，分别对应一级页表和二级页表。	39 位的虚拟地址通常被分为12位的页内偏移和三个9位的页号，分别对应一级页表、二级页表和三级页表。	48 位的虚拟地址通常被分为12位的页内偏移和四个9位的页号，分别对应一级页表、二级页表、三级页表和四级页表，支持 Sv48 的系统也应该支持 Sv39。

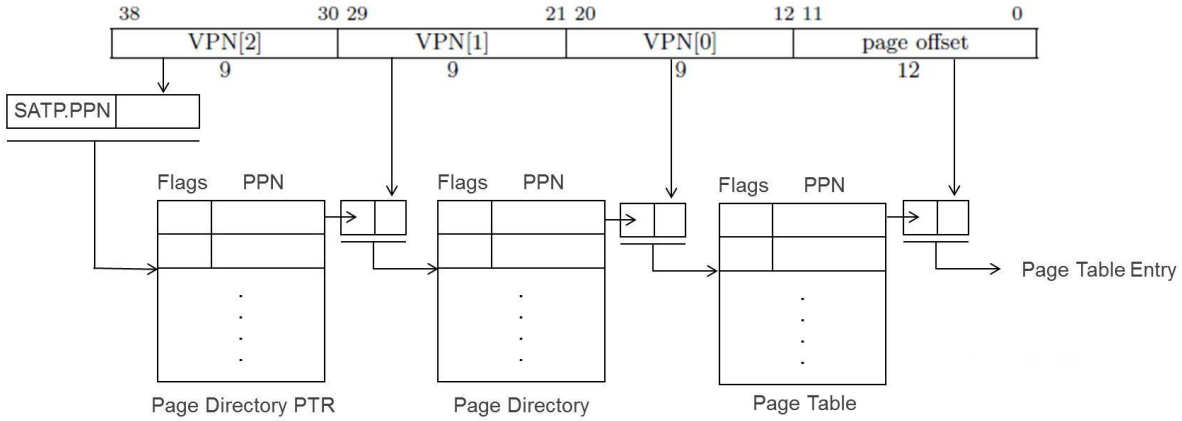
在`sv39`中，定义**物理地址(Physical Address)有 56位，而虚拟地址(Virtual Address) 有 39位。**该39位的分布如下。每一页占用4KB内存，页内使用虚拟地址低12位寻址（2^{12}）。虚拟地址的高二十七位划分为三级页号，每一级都有512个可用的页号。



下面介绍一下sv39的寻址过程。我们使用satp寄存器用于控制分页，satp的结构如下图示：



satp的MODE域用于控制开启关闭分页，以及选择将要使用的分页系统。PPN域保存根页表的物理地址，所有我们通过satp可以获取到一级页表的地址，再使用VPN[2]获取到对应的页表项。根据页表项可以取到二级页表的地址，再使用VPN[1]获取到二级页表项。然后继续用VPN[0]取出虚拟地址对应的物理页地址， 加上页内偏移地址，就完成了整个寻址过程，如下图所示。



因此`get_pte()`函数中的两段形式类似的代码，分别对应于两级映射。

第一级通过`PDX1`（宏）计算虚拟地址在根目录对应的页目录索引，并将`pdep1`指向VPN[1]；若不存在，进行分配。

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    //pgdir是页表基址(satp)，la是虚拟地址
    pde_t *pdep1 = &pgdir[PDX1(la)];
    if (!(*pdep1 & PTE_V)) { //传入的虚拟地址对应的页表项不存在
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {//不可以创建新的页表或没有成功分配新页表，则没有找到页表项
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);//获取该页面的起始物理地址
        memset(KADDR(pa), 0, PGSIZE);
        //现在在虚拟地址空间中 ，要转化为KADDR再memset
        //不管页表怎么构造，我们要确保物理地址和虚拟地址的偏移量始终相同，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); //创建页表项
    }
```

第二级将pdep1指向的页目录项中的物理地址转换位虚拟地址，进而转换成页目录项类型的指针，并通过`PDX0`（宏）计算虚拟地址在VPN[1]中的页目录索引；若不存在，进行分配。

从而获取到其在VPN[0]中的页表项，并返回对应页表项的地址。

```
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
//    pde_t *pdep0 = &((pde_t *)(PDE_ADDR(*pdep1)))[PDX0(la)];
    //和前面逻辑一样，页表不存在就分配一个
    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);
 //     memset(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（可能是刚分配好的）
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

- 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，我们认为这样的写法好，将查找和分配页表项功能合并在一个函数中，那么无论页表项是否存在，只需要调用一次这个函数就可以获得对应的页表项的指针，因为查找和分配是常常需要同时进行的操作，这样做可以简化代码，减少函数调用开销。但这么做会可能使得代码的可读性变差，若程序出现错误，调试的复杂度也将提高，可维护性差。在操作系统开发过程中，由于项目复杂程度高，参与人数多，因此代码的可维护性和可读性比较重要，因此在必要的时候可以将两功能单独拆分出来。

练习3：给未被映射的地址映射上物理页（需要编程）

> 补充完成do_pgfault（mm/vmm.c）函数，给未被映射的地址映射上物理页。设置访问权限的时候需要参考页面所在 VMA 的权限，同时需要注意映射物理页时需要操作内存控制结构所指定的页表，而不是内核的页表。
>
> 请在实验报告中简要说明你的设计实现过程。请回答如下问题：
>
> - 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。
> - 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？
>   - 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？

- 补充的代码如下：

```
        if (swap_init_ok) {
            struct Page *page = NULL;
            swap_in(mm,addr,&page); //在执行完后，page保存换入的物理页面，可能把内存原有的页面换出去
            page_insert(mm->pgdir,page,addr,perm);//更新页表，插入新的页表项，建立一个Page的phy addr与线性addr la的映射
            swap_map_swappable(mm,addr,page,1); //(3) make the page swappable.//标记这个页面将来是可以再换出来的
            page->pra_vaddr = addr;
        }
```

1. 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。

页目录项和页表项的结构类似，sv39 里面的一个页表项大小为 64 位 8 字节。其中第 53-10 位共44位为一个物理页号，表示这个虚拟页号映射到的物理页号，**为实现页替换算法提供了虚拟地址到物理地址的映射关系**；

后面的第 9-0 位共10位则描述映射的状态信息。页表项结构如下图所示：

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

映射状态信息各位的含义和在实现页替换算法中的作用：

- RSW：两位留给 S Mode 的应用程序，我们可以用来进行拓展，在实现页替换算法中使用。
- D：即 Dirty ，如果 D=1 表示自从上次 D 被清零后，有虚拟地址通过这个页表项进行写入。
- A，即 Accessed，如果 A=1 表示自从上次 A 被清零后，有虚拟地址通过这个页表项进行读、或者写、或者取指。

D和A可以运用到页替换算法中，使得操作系统得知当前页面是否有着较大的概率被访问。

- G，即 Global，如果 G=1 表示这个页表项是”全局"的，也就是所有的地址空间（所有的页表）都包含这一项
- U，即 user，U为 1 表示用户态 (U Mode)的程序 可以通过该页表项进映射。
- R,W,X 为许可位，分别表示是否可读 (Readable)，可写 (Writable)，可执行 (Executable)。
- V 表示这个页表项是否合法。如果为 0 表示不合法，此时页表项其他位的值都会被忽略

G、U、R、W、X、V为页替换算法提供了页面权限信息。

综上，页目录项和页表项中组成部分为ucore实现页替换算法提供了虚拟地址到物理地址的映射关系、对应页面的访问概率和权限。

2. 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?

   当我们出现了页访问异常，以下寄存器会被硬件自动设置，将一些信息提供给中断异常处理程序：

   - **sepc**，记录触发异常的那条指令的地址；
   - **scause**，记录异常发生的原因。
   - **stval**，记录一些异常处理所需要的辅助信息，比如访存、缺页异常，并把发生问题的目标地址或者出错的指令记录下来。

   然后保存上下文并执行**stvec**寄存器中指向的缺页中断程序，进行异常处理，处理完成后恢复上下文。

3. 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?

   有关系，数据结构page是最低级的页表，目录项是一级页表，存储的内容是页表项的起始地址（二级页表），而页表项是二级页表，存储的是每个页表的开始地址，这些内容之间的关系时通过线性地址高低位不同功能的寻址体现的。

练习4：补充完成Clock页替换算法（需要编程）

> 通过之前的练习，相信大家对FIFO的页面替换算法有了更深入的了解，现在请在我们给出的框架上，填写代码，实现 Clock页替换算法（mm/swap_clock.c）。(提示:要输出curr_ptr的值才能通过make grade)
>
> 请在实验报告中简要说明你的设计实现过程。请回答如下问题：
>
> - 比较Clock页替换算法和FIFO算法的不同。

FIFO页替换算法

FIFO算法每次选择最早进入内存的页面进行替换。通过把调入内存的页面根据调入的先后顺序排成一个队列，需要换出页面时选择队头页面即可。但当位进程分配的物理块数增大时，缺页次数会增加，称为belady现象。FIFO算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能较差。

clock页替换算法

clock算法为每个页面是指一个访问位，再将内存中的页面通过连接指针链接成一个循环队列，当某夜被访问时，将其访问位设置为1。当需要淘汰一个页面时，只需检查该页面的访问位。如果访问位为0，就将该页面换出；如果访问位为1.则将它置零，继续检查下一个页面，继续扫描其他页面，若扫描到0，则换出，若第一次扫描全部为1，则进行第二次扫描，重复上述过程，直到扫描到访问位为1的页面后，将其换出。本次实验使用上述clock算法。

简单的时钟置换算法仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过，就不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进型的时钟置换算法的思想。

练习5：阅读代码和实现手册，理解页表映射方式相关知识（思考题）

> 如果我们采用”一个大页“ 的页表映射方式，相比分级页表，有什么好处、优势，有什么坏处、风险？

- 优势：
  - 实现简单，仅需要维护一个页表即可。
  - 访问速度快，直接访问页表即可，中间无需其他查找操作。
  - 内存占用少，仅需要一个页表即可支持页表映射，所需内存少。
- 劣势：
  - 页表大小受限制，大虚拟内存空间不适用
    由于每个虚拟内存页对应一个页表项，当虚拟内存非常大，页表需要的物理内存随之增加，系统会进行限制。
  - 访问效率（另一个角度）较低
    页表是以线性结构存储页表项，虚拟空间很大时，页表项数量随之增大，查找速度慢，查找对应的物理页框也就慢了。
  - TLB缓存频繁缺失
    TLB缓存有限，虚拟空间很大，导致一级页表较大，导致TLB无法缓存最近使用的所有页表项，导致命中概率降低，查询转换速率降低。

涉及的知识点：

- Page Fault异常处理
- 页面置换机制