Lab3 缺页异常和页面置换

小组成员：张高2213219 张铭2211289 黄贝杰2210873

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？

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 kern/mm/swap_fifo.c
static int
_fifo_check_swap(void) {
    cprintf("write Virt Page c in fifo_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==4);
    cprintf("write Virt Page a in fifo_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==4);
    cprintf("write Virt Page d in fifo_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==4);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==4);
    cprintf("write Virt Page e in fifo_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    cprintf("write Virt Page a in fifo_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==6);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==7);
    cprintf("write Virt Page c in fifo_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==8);
    cprintf("write Virt Page d in fifo_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==9);
    cprintf("write Virt Page e in fifo_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==10);
```

```
    cprintf("write Virt Page a in fifo_check_swap\n");
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==11);
    return 0;
}
```

该函数主要是通过访问特定的虚拟内存地址（如 0x1000, 0x1010, 0x2000 等）来模拟对不同内存位置的写操作，如果某虚拟地址对应的页面不在内存中，则会发生 页面错误（page fault），并触发缺页中断来加载该页面，同时页面错误计数器pgfault_num会增加。这种方式是用来触发处理页面错误的相关函数，来验证 FIFO 页面置换算法和页面错误处理机制是否工作正常。

```
///2 kern/trap/trap.c
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");
}
```

pgfault_handler函数是操作系统内核中用来处理页面错误（page fault）的函数。页面错误是指在程序运行时，尝试访问未映射的内存页面，或者访问了不存在或不可访问的内存页时，会触发一个异常或中断。操作系统通过捕获这个异常来执行相关的处理，，例如分配新的内存页或替换新的内存页面。

```
///3 kern/mm/vmm.c
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    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 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);

    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;
            swap_in(mm, addr, &page);
            page_insert(mm->pgdir, page, addr, perm);
            swap_map_swappable(mm, addr, page, 1);
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
   }

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

do_pgfault函数具体处理页面错误（Page Fault）。当程序访问一个不在物理内存中的虚拟地址时，操作系统需要处理页面错误并加载相关的页面。这个函数在发生页面错误时执行，尝试找到相应的虚拟内存区域（VMA），分配页面，并将其映射到虚拟地址空间。总体流程是：

- 查找当前虚拟地址所在的虚拟内存区域（VMA）。find_vma函数用于根据虚拟地址addr查找相应的虚拟内存区域（VMA）。
- 检查该地址是否合法，如果不合法，则返回错误。
- 如果该地址尚未分配物理页面，尝试分配新页面并更新页表。调用get_pte函数检查当前页表（Page Table）中是否已有对应的页面表项（PTE）。
- 如果该地址已经存在并且启用了交换机制，则从交换空间加载页面。swap_map_swappable函数将页面标记为可交换出，并将页面的虚拟地址设置为当前访问的地址。
- 返回处理结果（成功或失败）。

```
///4 kern/mm/vmm.c
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;
}
```

find_vma函数用于在给定的虚拟内存结构体mm_struct中查找一个虚拟内存区域（vma_struct），这个区域包含了给定的虚拟地址addr。它通过遍历mm_struct中的虚拟内存区域链表来查找包含该地址的虚拟内存区域，并缓存最近访问的区域以提高性能。

```
///5 kern/mm/pmm.c
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);
        *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)];
    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);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

get_pte函数的作用是根据给定的虚拟地址la查找或创建对应的页面表项。它通过遍历页目录和页表（分为两级），检查是否存在相应的页表项。如果不存在且create参数为真，则会分配新的物理页面，并更新相应的页目录和页表项。该函数在操作系统的虚拟内存管理中非常重要，主要用于映射虚拟地址到物理地址，并保证内存的正确分配与访问。

```
///6 kern/mm/swap.c
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);
     // cprintf("SWAP: load ptep %x swap entry %d to vaddr 0x%08x, page %x,
No %d\n", ptep, (*ptep)>>8, addr, result, (result-pages));

     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;
}
```

swap_in函数是操作系统内核中用于将页面从交换空间（swap space）加载到物理内存的函数,交换空间是指将物理内存的某些页面写到磁盘上，以便腾出内存供其他页面使用。当需要访问某个页面，但该页面已经被换出到磁盘时，操作系统将磁盘上的页面加载回内存，可能还会触发页面交换。

具体来说，swap_in函数会分配一个内存页，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。

```
///7 kern/mm/pmm.h
#define alloc_page() alloc_pages(1)
```

该宏将文本alloc_page()替换为函数调用alloc_pages(1)，即分配一个物理页。alloc_page()在swap_in函数出现，为磁盘上的页面加载回内存而分配一个物理页。

```
///8 kern/mm/pmm.c
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);

        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        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;
}
```

alloc_pages函数的目的是分配物理内存页。它首先尝试直接分配所需数量的页面，如果分配失败，且只要求分配一个页面时，会通过调用swap_out将部分页面交换到磁盘，腾出空间来进行新的内存分配。这样，如果系统内存紧张，它会通过页面交换（swap）来增加内存的可用空间。

```
///9 kern/mm/default_pmm.c
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

default_alloc_pages函数实现了分配物理内存页的具体方式，目的是从空闲页面链表中分配指定数量的物理页面。具体来说，它会查找一块足够大的空闲页面并返回；如果没有足够的空闲页面，函数会返回 NULL。

```
///10 kern/mm/swap.c
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);
          if (r != 0) {
                    cprintf("i %d, swap_out: call swap_out_victim
failed\n",i);
                  break;
          }
          //assert(!PageReserved(page));

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

          v=page->pra_vaddr;
          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) {
                    cprintf("SWAP: failed to save\n");
                    sm->map_swappable(mm, v, page, 0);
                    continue;
          }
          else {
                    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);
          }

          tlb_invalidate(mm->pgdir, v);
     }
     return i;
}
```

swap_out函数用于将 n 个页面从内存交换到磁盘的交换空间。它通过以下步骤进行：

- 选择一个页面进行交换。调用swap_out_victim函数来选择一个页面进行交换。此函数会选择当前进程mm的一个页面，并将其地址保存在page中。
- 将该页面的数据写入交换空间。swapfs_write函数将页面的数据写入交换空间，如果写入失败，调用map_swappable函数将页面恢复为可交换的状态，继续进行下一次循环。

- 更新该页面的页表项，将其标记为在交换空间中的位置。
- 释放内存中的页面。`free_page`函数释放页面`page`，因为它已经被写入到交换空间中，不再需要占用物理内存。
- 使相关的 TLB 条目失效，以确保下一次访问该页面时能重新加载。`tlb_invalidate`函数使相关的 TLB（Translation Lookaside Buffer，快速页表缓存）条目失效，因为该页面已经被换出到交换空间。

这个过程是在操作系统内存管理中处理内存不足时进行页面交换的一部分。它确保了操作系统能够有效地利用有限的物理内存，通过交换机制实现虚拟内存的扩展。

```
///11 kern/mm/swap_fifo.c
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;
}
```

`_fifo_swap_out_victim`函数实现了 FIFO 算法，将最先进入内存的页面优先置换出去，也就是最早加载到内存的页面。其简单地从页面链表的头部选择页面进行置换，它会从 `pra_list_head`队列中删除第一个页面（最早加载的页面），并将该页面的指针返回给`ptr_page`。

```
///12 kern/mm/swap_fifo.c
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

    //(1)link the most recent arrival page at the back of the pra_list_head
qeueue.
    list_add(head, entry);
```

```
    return 0;
}
```

_fifo_map_swappable函数根据 FIFO 页面置换算法，将新的页面（或交换回来的页面）添加到页面链表的末尾，表示它是最近加载的页面。

```
///13 kern/mm/pmm.c
void tlb_invalidate(pde_t *pgdir, uintptr_t la) { flush_tlb(); }

/// kern/mm/pmm.h
static inline void flush_tlb() { asm volatile("sfence.vma"); }
```

tlb_invalidate函数调用flush_tlb函数，而flush_tlb是一个内联函数，它会调用sfence.vma汇编指令，刷新 TLB，确保虚拟地址到物理地址的映射是最新的。该操作通常在操作系统需要更新页表、映射变化或切换进程时使用，以防止访问过期的 TLB 条目。

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？

相似部分的代码如下:

```
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);

    *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)];
    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);
    }
```

第一段代码处理的是最高级别的页表项（PDX1），而第二段代码处理的是次一级的页表项（PDX0）。尽管它们处理的是不同级别的页表项，但基本的操作逻辑是相同的：获取对应页、不存在则创建、设置页属性及其内存状态、构造页表项。

我认为挺好的

首先查找虚拟地址对应页表项时确实有可能不存在页表项，所以每次都会有一个缺失检

查，如果缺失则会调用相应的创建函数。

其次因为**代码的简洁性和复用性**：合并查找和分配功能可以减少代码的重复，使得函数更加简洁。这样的设计使得调用者不需要分别处理查找和分配的逻辑，从而简化了调用者的代码。

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。

首先，查找包含给定逻辑地址addr的虚拟内存区域，在当前进程的内存管理结构mm中找到对应的VMA（虚拟内存区域）

```
//try to find a vma which include addr
struct vma_struct *vma = find_vma(mm, addr);
```

判断VMA是否存在以及addr是否在VMA的范围内，如果不满足，则输出错误信息并返回。

```
//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;
}
```

设置权限

- `uint32_t perm = PTE_U;`：初始化权限为用户可访问。
- 如果VMA的 `vm_flags` 包含 `VM_WRITE`，则设置权限为可读写（`perm |= (PTE_R | PTE_W);`）。

```
/* 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);

}
```

将addr按页大小PGSIZE进行向下对齐。

```
addr = ROUNDDOWN(addr, PGSIZE);
```

尝试获取页表项pte，如果下一级页表不存在则分配一个物理页来建立新页表，并建立页表项以及逻辑地

址addr的映射关系。

```
 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.
```

如果pte已经存在，说明该页表项是一个交换条目，需要从磁盘加载数据到物理页，并建立映射关系，并设置页面可交换。

```
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);
        //(2) According to the mm,
        //addr AND page, setup the
        //map of phy addr <--->
        //logical addr
        page_insert(mm->pgdir, page, addr, perm);
        //(3) make the page swappable.
        swap_map_swappable(mm, addr, page, 1);

        page->pra_vaddr = addr;
      } else {
        cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
        goto failed;
      }
    }
```

关键的三步(也是实验要求补齐的代码):

`swap_in(mm, addr, &page);`：从磁盘加载数据到内存页面。

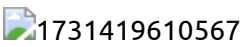
`page_insert(mm->pgdir, page, addr, perm);`：设置物理地址和逻辑地址的映射。

`swap_map_swappable(mm, addr, page, 1);`：设置页面为可交换。

成功返回0，失败返回错误信息。

- **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。

在sv39中，页表项的结构如下：

 1731419610567

页目录项（PDE）和页表项（PTE）在ucore实现页替换算法中扮演着核心角色。它们的主要组成部分及其潜在用处如下：

- **存在位（P）**：指示页表或页是否在物理内存中。在页替换算法中，这可以帮助识别哪些页面当前被加载到内存中，哪些不在
- **读写位（R/W）：**指示页面是否可写。在页替换算法中，这有助于确定页面是否被修改过（脏页），因为这将影响是否需要在换出时写回磁盘
- **用户/超级用户位（U/S）**：指示页面是否可以被用户模式访问。这在页替换算法中用于确保安全性，防止用户程序访问或修改内核空间。
- **访问位（A）**：指示页面是否被访问过。页替换算法可以使用这个位来识别最近被访问的页面，这对于某些算法（如LRU）来说很重要
- **脏位（Dirty）**：表示页面自从加载到内存后是否被修改过。这对于页替换算法中的页面换出策略至关重要，因为脏页需要在换出前写回磁盘
- **问题**：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？

代码层面：如果出现页面访问异常，程序会进入 trap/trapentry.S 当中，保存上下文，再进入

trap.c 文件当中进行异常处理，处理结束后，再逐步退回至发生异常的代码处，重新执行。

在 trap.c 文件当中：会从 pgfault_handler 函数中执行 do_pgfault 函数。

在 pg_fault 函数当中，根据发生页面访问异常的地址 addr 寻找到对应的 vma 。获取 vma 的权限

后，根据 addr 对齐后的地址，寻找相应的 page_table_entry 。如果没有找到，就立马创建一个

pte 。找到 pte 后，我们将该页面换入，存入物理地址和虚拟地址的映射，将该页面设置为可被替

换的。

当ucore的缺页服务例程在执行过程中访问内存出现页访问异常时，硬件会执行以下操作：

1. 将异常类型设置为页访问异常
2. 保存当前的程序状态/上下文（如PC、寄存器等）
3. 转移控制权到操作系统的缺页异常处理程序
4. 恢复上下文
5. 继续执行当前缺页异常处理程序

- **问题**：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？

当然有的，我们首先看Page结构体的组成：


1731420476216

其中的pra_vaddr就是这个页所对应的虚拟地址。根据这个地址，我们就可以知道其对应的页

目录项和页表项。

Page结构：每个Page结构代表一个物理页帧，记录了页面的状态信息，如是否被分配、是否被修改（脏位）、是否被访问等

页表项（PTE）：Page结构中的某些字段（如`pra_vaddr`）用于记录页面对应的虚拟地址，这与页表项中的虚拟地址字段相对应。页表项中的物理地址字段（通常是页帧号）与Page结构中记录的物理页帧号相对应

页目录项（PDE）：页目录项指向页表的物理地址，而Page结构中的页表项（PTE）是页表中的一个条目，它们共同构成了虚拟地址到物理地址的映射关系

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)

实现过程:

完成以下各个函数：

- **_clock_init_mm**

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: YOUR CODE*/
     // 初始化pra_list_head为空链表
     // 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
     // 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
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     list_init(&pra_list_head);
     curr_ptr = &pra_list_head;
     mm->sm_priv = &pra_list_head;
     return 0;
}
```

初始化pra_list_head为空链表，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，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。

- **_clock_map_swappable**

```
static int
_clock_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 && curr_ptr != NULL && head != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // link the most recent arrival page at the back of the pra_list_head
qeueue.
    // 将页面page插入到页面链表pra_list_head的末尾
    // 将页面的visited标志置为1，表示该页面已被访问
    list_add(head, entry);
```

```
    page->visited = 1;
    return 0;
}
```

将页面page插入到页面链表pra_list_head的末尾，即将新的页面添加到clock链表（环形链表）尾部。

- **_clock_swap_out_victim**

```
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);
     /* 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
    while (1) {
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 编写代码
        // 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
        curr_ptr = list_next(curr_ptr);
        if(curr_ptr == head) {
            curr_ptr = list_next(curr_ptr);
        }
        struct Page *page = le2page(curr_ptr, pra_page_link);
        if(!page->visited) {
            cprintf("curr_ptr 0xffffffff%x\n", curr_ptr);
            list_del(curr_ptr);
            *ptr_page = page;//将该页面指针赋值给ptr_page作为换出页面
            break;
        } else {
            page->visited = 0;
        }
    return 0;
}
```

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`结构体指针。

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`置为0，表示该页面已被重新访问。

Clock算法的核心部分是页面替换策略。当需要换出页面时，从当前指针`curr_ptr`开始向前遍历页面链表(环形链表)，由于`head`指针无法利用`le2page`宏转成`Page`结构体指针，因此先进行判断。检查页面的访问位

visited，如果访问位为1，则将其清零，并移动到上一个页面。否则，选择该页面为替换页面，将其从链表中移除，并返回。

比较 Clock 页替换算法和 FIFO 算法的不同。

代码实现方面

- 对于_clock_init_mm函数，FIFO算法初始化mm_struct结构体中的页面替换链表，用于后续的页面替换算法操作，Clock算法在此基础上引入curr_ptr指针指向pra_list_head，表示当前页面替换位置为链表头，用于遍历链表的操作。
- 对于_clock_map_swappable函数，FIFO算法将新的页面（或交换回来的页面）添加到页面链表的末尾，表示它是最近加载的页面。，Clock算法在此基础上将页面的visited标志置为1，表示该页面已被访问。
- 对于_clock_swap_out_victim函数，FIFO算法简单地从页面链表的头部选择页面进行置换。它会从pra_list_head队列中删除第一个页面（最早加载的页面），并将该页面的指针返回给ptr_page，Clock算法遍历页面链表，检查每个页面的visited标志，如果页面被访问过即visited为1，就将其visited重置为0；如果页面没有被访问过即visited为0，就选择该页面并将其从链表中删除。

总体来看

- 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。
- 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。
- *Clock页面置换算法通过使用访问位来改进FIFO算法，减少了Belady异常的发生几率。Clock页面置换算法使得频繁访问的页面可以保留在内存中，而较少访问的页面将被优先替换掉。*

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？

一级页表：

- 优点

实现简单，只需要维护一个表即可！

访问速度快，直接访问页表即可，无需其他的查找操作

提高TLB命中率

- 缺点

1. 如果虚拟内存非常大，页表所需要的物理内存也会变大。
2. 如果页表中的页表项数量增多，查找页表项就会很慢，访问相对而言就会变慢

分级页表：

- 优点

适用于大内存空间，引入多级页表可以将页表项分散到多个页表中存储

- 缺点

1. 需要维护多个表
2. 访问存时，需要访问多个表才能最终访问到物理地址耗时
3. 如果页表项较少，不太适用分级页表，反而会占用更多的内存。

扩展练习Challenge：实现不考虑实现开销和效率的LRU页替换算法（需要编程）

实现思路

我们可以修改 `pra_list_head`的组织形式。在`fifo置换算法`中，页面会按照第一次被访问的顺序插入到`pra_list_head`中， 为了实现LRU算法，我们可以通过检测页面的访问情况，每当有一个页面被访问时，将其插入到链表的首部， 这样我们就可以保证链表的首部始终为最近访问的页，而尾部为最长时间未被访问的页。需要发生页面置换时，直接将尾部页面置换。

实现以上思路的关键在于如何`检测到页面的访问情况`，我们可以通过触发`pgfault`来实现这一目的。具体方式为：

- 在访问页面之前，将所有页面的页表项权限设置为不可读。因此在访问一个页面时，会引发缺页异常。
- 在处理缺页异常时，更新该页面的权限为可读，并将其插入链表头部。
- 每当发生缺页异常时，更新相关页面的权限为可读，并把该页面插入链表头部。

具体实现

添加`swap_LRU_vaild`变量

由于以上思路中使用LRU时会更改之前pgfault的处理方式，因此我们在`swap.c`中增加一个布尔变量`swap_LRU_vaild`

- 启用LRU的方式如下：在`swap_init`函数中将`swap_LRU_vaild`设置为`true`并将`sm`为设置为`swap_manager_LRU`

```
int swap_init(void)
{
    //其他剩余代码...
```

```
    swap_lru_vaild = true;
    sm = &swap_manager_LRU;
    //其他剩余代码...
}
```

swap_LRU核心函数

- 创建swap_LRU.c和swap_LRU.h，并定义与fifo相同的函数接口， 核心函数为_LRU_init_mm、_LRU_map_swappable和_LRU_swap_out_victim， 分别用于LRU初始化、获取可进行LRU置换的页面以及选择要进行LRU置换的最久未使用的页面（即链表末尾的页面），代码如下所示：

```
static int
_LRU_init_mm(struct mm_struct *mm)
{
    // 初始化LRU链表的头节点
    list_init(&pra_list_head);
    // 将mm结构体中的sm_priv字段指向LRU链表的头节点
    mm->sm_priv = &pra_list_head;
    // 初始化成功，返回0
    return 0;
}

static int
_LRU_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
int swap_in)
{
    // 获取LRU链表的头节点
    list_entry_t *head = (list_entry_t*) mm->sm_priv;
    // 获取该页面的链表节点
    list_entry_t *entry = &(page->pra_page_link);
    // 确保entry和head不为空
    assert(entry != NULL && head != NULL);
    // 将该页面的链表节点添加到LRU链表的头部
    list_add((list_entry_t*) mm->sm_priv, entry);
    // 成功执行，返回0
    return 0;
}

static int
_LRU_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int
in_tick)
{
    // 获取LRU链表的头节点
    list_entry_t *head = (list_entry_t*) mm->sm_priv;
    // 确保head不为空
    assert(head != NULL);
    // 确保in_tick为0（意味着当前不是在时钟中断中执行）
    assert(in_tick == 0);
    // 获取LRU链表的前一个元素（即最久未使用的页面）
    list_entry_t* entry = list_prev(head);
    // 如果链表中有元素（entry != head），则执行页面交换
```

```
    if (entry != head) {
        // 从链表中删除该元素
        list_del(entry);
        // 将ptr_page指向该页面结构体
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        // 如果链表为空，则将ptr_page置为NULL
        *ptr_page = NULL;
    }
    // 成功执行，返回0
    return 0;
}
```

LRU的pgfault处理方式

我们可以在vmm.c的do_pgfault函数中添加对LRU的特殊处理，代码如下：

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr)
{
    // 检查是否需要进行LRU（最近最少使用）页面交换操作
    if (swap_lru_vaild) {
        // 定义一个临时指针，表示某个页面的页表项
        pte_t* temp = NULL;
        // 获取页表项，地址是`addr`，标志位为0
        temp = get_pte(mm->pgdir, addr, 0);
        // 如果页表项有效且该页表项包含有效位（PTE_V）和可读位（PTE_R），表示该页是有
效且可访问的
        if (temp != NULL && (*temp & (PTE_V | PTE_R))) {
            // 输出调试信息，表示发生了LRU页面缺页
            cprintf("lru page fault at 0x%x\n", addr);
            // 如果初始化的交换机制已经启动
            if (swap_init_ok) {
                // 获取LRU链表的头部，即`sm_priv`字段
                list_entry_t *head = (list_entry_t*) mm->sm_priv, *le =
head;
                // 遍历LRU链表
                while ((le = list_prev(le)) != head) {
                    // 获取当前页面对象
                    struct Page* page = le2page(le, pra_page_link);
                    // 获取该页面的页表项
                    pte_t* ptep = get_pte(mm->pgdir, page->pra_vaddr, 0);
                    // 如果页表项有效，将其可读位（PTE_R）清除
                    *ptep &= ~PTE_R;
                }
            }
            // 获取当前请求页的页表项
            pte_t* ptep = NULL;
            ptep = get_pte(mm->pgdir, addr, 0);
            // 设置该页面的可读位（PTE_R），表示当前页面已被访问
            *ptep |= PTE_R;
```

```
            // 如果交换机制没有初始化好，直接返回0
            if (!swap_init_ok)
                return 0;
            // 将页表项转换为页面对象
            struct Page* page = pte2page(*ptep);
            // 获取LRU链表的头部
            list_entry_t *head = (list_entry_t*) mm->sm_priv, *le = head;
            // 遍历LRU链表
            while ((le = list_prev(le)) != head) {
                // 获取当前页面对象
                struct Page* currPage = le2page(le, pra_page_link);
                // 如果当前页面就是目标页面，进行链表操作
                if (page == currPage) {
                    // 从链表中删除该页面
                    list_del(le);
                    // 将该页面重新添加到链表头部
                    list_add(head, le);
                    break;
                }
            }
            return 0;
        }
    }
    //其他剩余代码...
}
```

测试结果

在命令行执行`make qemu`指令，输出结果如下：

```
SWAP: manager = LRU swap manager
BEGIN check_swap: count 2, total 31661
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
Store/AMO page fault
page fault at 0x00001000: K/W
Store/AMO page fault
page fault at 0x00002000: K/W
Store/AMO page fault
page fault at 0x00003000: K/W
Store/AMO page fault
page fault at 0x00004000: K/W
set up init env for check_swap over!
write Virt Page c in LRU_check_swap
write Virt Page a in LRU_check_swap
write Virt Page d in LRU_check_swap
write Virt Page b in LRU_check_swap
write Virt Page e in LRU_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
```

```
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in LRU_check_swap
write Virt Page a in LRU_check_swap
Store/AMO page fault
page fault at 0x00001000: K/W
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in LRU_check_swap
Store/AMO page fault
page fault at 0x00002000: K/W
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in LRU_check_swap
Store/AMO page fault
page fault at 0x00003000: K/W
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in LRU_check_swap
Store/AMO page fault
page fault at 0x00004000: K/W
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in LRU_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in LRU_check_swap
Load page fault
page fault at 0x00001000: K/R
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 1, total is 8
check_swap() succeeded!
```