lab3_team

流程

pmm_init函数完成物理内存的管理初始化

执行中断和异常相关的初始化工作。

此过程涉及调用pic_init函数和idt_init函数，用于初始化处理器中断控制器（PIC）和中断描述符表（IDT）

调用vmm_init函数进行虚拟内存管理机制的初始化。

在此阶段，主要是建立虚拟地址到物理地址的映射关系，为虚拟内存提供管理支持。继续执行初始化过程

调用ide_init函数完成对用于页面换入和换出的硬盘（通常称为swap硬盘）的初始化工作。

在这个阶段，ucore准备好了对硬盘数据块的读写操作，以便后续页面置换算法的实现。

其实ide_init函数什么也没做，只是一个空函数，但其中定义了几个比较有用的函数，结合源码，为方便理解，我们给出了一些自己的注释信息

```
///in kern/driver/ide.c

void ide_init(void) {}//该函数被定义为空，即不执行任何操作。通常，这个函数可以用于初始化

#define MAX_IDE 2  //定义了MAX_IDE的值为2，表示IDE硬盘的数量上限为2。
#define MAX_DISK_NSECS 56  //表示每个IDE硬盘最多有56个扇区
static char ide[MAX_DISK_NSECS * SECTSIZE]; //静态字符数组，用于模拟IDE硬盘的存储空间
                                            //数组的大小为MAX_DISK_NSECS * SECTSI
                                            //每个扇区的大小为SECTSIZE字节。

bool ide_device_valid(unsigned short ideno) { return ideno < MAX_IDE; }

size_t ide_device_size(unsigned short ideno) { return MAX_DISK_NSECS; }

int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
                  size_t nsecs) {
    int iobase = secno * SECTSIZE;
```

```
    memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
    return 0;

    //用于从指定IDE设备读取扇区数据。
    //接受设备号ideno、起始扇区号secno、目标缓冲区指针dst，以及要读取的扇区数nsecs。
    //函数会将扇区数据复制到目标缓冲区，并返回一个整数值，用于表示操作是否成功。

}

int ide_write_secs(unsigned short ideno, uint32_t secno, const void *src,
                   size_t nsecs) {
    int iobase = secno * SECTSIZE;
    memcpy(&ide[iobase], src, nsecs * SECTSIZE);
    return 0;

    //用于向指定IDE设备写入扇区数据。
    //接受设备号ideno、起始扇区号secno、源数据缓冲区指针src,以及要写入的扇区数nsecs。
    //函数会将源数据复制到IDE硬盘的模拟存储空间中，并返回一个整数值，用于表示操作是否成功。
}

```

swap_init函数用于初始化页面置换算法，这其中包括Clock页替换算法的相关数据结构和初始化步骤。

通过swap_init，ucore确保页面置换算法准备就绪，可以在需要时执行页面换入和换出操作，以优化内存的利用。

在swap_init中，首先调用了swapfs_init函数，之后使用了时钟算法作为交换管理器。

```
//in kern/fs/swapfs.c

void
swapfs_init(void) {
    static_assert((PGSIZE % SECTSIZE) == 0);
    if (!ide_device_valid(SWAP_DEV_NO)) {
        panic("swap fs isn't available.\n");
    }
    max_swap_offset = ide_device_size(SWAP_DEV_NO) / (PGSIZE / SECTSIZE);

    //使用static_assert进行断言检查，确保页的大小能够整除扇区的大小。
```

```
    //检查交换分区设备是否有效，如果无效则调用panic函数触发内核崩溃。


}

//in kern/mm/swap.c

int
swap_init(void)
{
     swapfs_init();

     // Since the IDE is faked, it can only store 7 pages at most to pass the te
     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 Algorith
     //将sm指针指向名为swap_manager_clock的交换管理器结构体实例。这里选择了时钟算法作为
     int r = sm->init();

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

     return r;
}
```

练习1：理解基于FIFO的页面替换算法（思考题）

换入

在vmm.c的`do_pafault`的缺页处理函数中，会进行页面的换入操作：

```
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);
    //查找缺失地址是否在当前mm的vma_list中
```

```
 7
 8      pgfault_num++;
 9      //If the addr is in the range of a mm's vma?
10      if (vma == NULL || vma->vm_start > addr) {
11          cprintf("not valid addr %x, and  can not find it in vma\n", addr);
12          goto failed;
13      }
14
15      uint32_t perm = PTE_U;
16      if (vma->vm_flags & VM_WRITE) {//如果这段虚拟地址属于writable，则perm会增加Read
17          perm |= (PTE_R | PTE_W);
18      }
19      addr = ROUNDDOWN(addr, PGSIZE); //向下对齐到addr所在的页
20
21      ret = -E_NO_MEM;
22
23      pte_t *ptep=NULL;
24      ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
25                                           //PT(Page Table) isn't existed, then
26                                           //create a PT.
27      if (*ptep == 0) {    //如果该一级页表项记录为0，代表最终映射的物理页面没有分配
28          if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) { //分配物理页面
29              cprintf("pgdir_alloc_page in do_pgfault failed\n");
30              goto failed;
31          }
32      } else {
33          if (swap_init_ok) {
34              struct Page *page = NULL;
35              if(swap_in(mm,addr,&page))  //将addr所在的页swap到page对应的虚拟页中，m
36              {
37                  cprintf("swap in failed!\n");
38                  goto failed;
39              };
40              page_insert(mm->pgdir,page,addr,perm);  //建立addr所在的地址与page的映
41              swap_map_swappable(mm,addr,page,1);     //加入换入换出队列
42
43              page->pra_vaddr = addr;                 //该page对应的地址
44          } else {
45              cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
46              goto failed;
47          }
48     }
49
50     ret = 0;
51 failed:
52      return ret;
53 }
```

在这个流程中，直接或间接地调用了一些重要的函数和宏定义：

1. **find_vma**:用于在传入的mm中查找是否有包含了addr的一段vma，如果缺失这一步骤，可能导致内存管理混乱，将不属于这一PDT的地址换入

```
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;   //mmap_cache 存储最近访问过的vma，利用空间局部性
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
                bool found = 0;
                list_entry_t *list = &(mm->mmap_list), *le = list;  //从vma的
                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;   //update cache
        }
    }
    return vma;
}

```

2. **ROUNDDOWN**宏定义：用于将addr向下对齐至页面开始处，方便之后的换入
3. **get_pte**:用于得到addr地址所在的page对应的一级页表项

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    pde_t *pdep1 = &pgdir[PDX1(la)];   //PDX1 三级页表号
    if (!(*pdep1 & PTE_V)) {        //三级页表项的valid位为0，无法访问或不存在
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);  //pa是page结构体对应的物理地址
```

```
        memset(KADDR(pa), 0, PGSIZE);  //KADDR(pa) == page2kva(page)，从物理地址得
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);  //建立页表项
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)]; //得到二级页表项
    if (!(*pdep0 & PTE_V)) { //和以上类似，如果该页不存在，则创建页和页表项
        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)]; //最终得到一级页表项
}
```

3. 如果由get_pte得到的一级页表项的内容为0，代表最终的物理页面还没有分配，此时要调用**pgdir_alloc_page**,其作用是找到传入的PDT下la所对应的一级页表项pte，并通过分配一个page结构体，将page所对应地址与pte建立映射。总而言之，其作用是分配一个最终内存页，并和一个一级页表项建立映射：

```
// pgdir_alloc_page - call alloc_page & page_insert functions to
//                  - allocate a page size memory & setup an addr map
//                  - pa<->la with linear address la and the PDT pgdir
struct Page *pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page(); //分配一个页
    if (page != NULL) {
        if (page_insert(pgdir, page, la, perm) != 0) {
            free_page(page);
            return NULL;
        }
        if (swap_init_ok) {
            swap_map_swappable(check_mm_struct, la, page, 0);
            page->pra_vaddr = la;
            assert(page_ref(page) == 1);
            // cprintf("get No. %d  page: pra_vaddr %x, pra_link.prev %x,
            // pra_link_next %x in pgdir_alloc_page\n", (page-pages),
            // page->pra_vaddr,page->pra_page_link.prev,
            // page->pra_page_link.next);
        }
    }

    return page;
```

```
}
```

在pdir_alloc_page中，又有几个函数需要进一步理解：

4. **page_insert**:用于建立page所对应的物理内存地址和pdir下la所对应的一级页表项的映射，简短理解，就是建立一个一级页表项到物理地址的映射

```
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    pte_t *ptep = get_pte(pgdir, la, 1); //找到该一级页表项
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page); //该页的引用加1
    if (*ptep & PTE_V) { //valid，说明页表项已经存在且有映射
        struct Page *p = pte2page(*ptep); //找到映射的页
        if (p == page) { //恰好为同一页，则简单删去一个引用即可
            page_ref_dec(page);
        } else { //不是同一页，则将之前的映射抹去
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = pte_create(page2ppn(page), PTE_V | perm); //建立映射，将page对应的物理
    tlb_invalidate(pgdir, la); //刷新tlb
    return 0;
}
```

其中又有几个宏定义和函数：

5. **pte2page**:从一个pte，得到其对应的物理地址所对应的page结构体

```
static inline struct Page *
pte2page(pte_t pte) {
    if (!(pte & PTE_V)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte)); //实际上是对PTE_ADDR和pa2page的封装
}
#define PTE_ADDR(pte)   (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT - PTE_PPN_SHIF
//得到pte中存储的物理地址

static inline struct Page *
pa2page(uintptr_t pa) { //由物理地址转换，得到page结构体
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
```

```
    }
    return &pages[PPN(pa) - nbase];
}
```

6. **page_remove_pte**:删去pte到其之前对应的page之间的映射

```
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
        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 second page table entry
        tlb_invalidate(pgdir, la);  //(6) flush tlb
    }
}
```

7. **pte_create**:根据传入的物理页号和保留位设置，建立一个pte页表项：

```
static inline pte_t pte_create(uintptr_t ppn, int type) {
  return (ppn << PTE_PPN_SHIFT) | PTE_V | type;
}
```

8. **swap_map_swappable**:实际上是对sm替换管理器的map_swappable的封装，作用是将某一页加入到替换队列中

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

9. 在整个替换流程中，会调用**swap_in**函数进行页面的换入，其流程大致如下：

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();//分配新的一页，用于装载换入的数据
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);
     //得到addr所在的page对应的一级页表项

     int r;
     if ((r = swapfs_read((*ptep), result)) != 0) //读取页，并判断是否读取成功
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n", (
     *ptr_result=result; //记录结果。即换入数据后的page
     return 0;
}
```

其中有一些可以进一步分析解读的函数：

10. **swapfs_read**：从硬盘文件fs中读取换入的页，可见该函数实际上是对ide_read_secs的封装，其作用是，将entry指向的硬盘处的数据，写入page所指向的虚拟地址中：

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

有几个宏定义函数可以进一步解释：

11. swap_offset:将一个swap_entry右移八位，得到其记录的硬盘偏移量：

```
/* *
 * swap_entry_t
 * --------------------------------------------
 * |         offset        |   reserved   | 0 |
 * --------------------------------------------
 *           24 bits            7 bits    1 bit
 * */
#define swap_offset(entry) ({                                       \
               size_t __offset = (entry >> 8);                        \
               if (!(__offset > 0 && __offset < max_swap_offset)) {     \
                    panic("invalid swap_entry_t = %08x.\n", entry);     \
```

```
12              }                                                   \
13              __offset;                                           \
14          })
```

12. page2kva:从结构体page转换至其对应的虚拟地址，其步骤是，从结构体page得到其物理页号，再将物理页号左移12位得到物理地址，再将物理地址加上va_pa_offset偏移量得到虚拟地址

```
1  static inline void *
2  page2kva(struct Page *page) {
3      return KADDR(page2pa(page));
4  }
5
6  static inline uintptr_t
7  page2pa(struct Page *page) {
8      return page2ppn(page) << PGSHIFT;
9  }
10
11 static inline ppn_t
12 page2ppn(struct Page *page) {
13     return page - pages + nbase;
14 }
15
16 #define KADDR(pa)                                                   \
17     ({                                                              \
18         uintptr_t __m_pa = (pa);                                    \
19         size_t __m_ppn = PPN(__m_pa);                               \
20         if (__m_ppn >= npage) {                                     \
21             panic("KADDR called with invalid pa %08lx", __m_pa);    \
22         }                                                           \
23         (void *)(__m_pa + va_pa_offset);                            \
24     })
```

13. 对于ide_read_secs，其作用则是，从第secno块扇区，复制nsecs块扇区到（即读到）dst中：

```
1  int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
2                    size_t nsecs) {//项目代码中只挂载一块硬盘，因此ideno不起作用
3      int iobase = secno * SECTSIZE;
4      memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
5      return 0;
6  }
```

换出

在swap.c中，可以得到换出页面时的操作函数swap_out:

```
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); //提取出换出页
          if (r != 0) {
                    cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                  break;
          }

          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 sw
                    *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
                    free_page(page);
          }

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

在换出阶段，主要利用 `swap_out_victim` 函数找到替换页并进行换出，在此处将最先进入队列的page取出，用于替换，并用ptr_page来记录这个page：

```
1
2  static int
3  _fifo_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick
4  {
5       list_entry_t *head=(list_entry_t*) mm->sm_priv; //得到当前的Giga page的页链表
6           assert(head != NULL);
7       assert(in_tick==0);
8      list_entry_t* entry = list_prev(head);
9      //由于新加入的页总添加至head之后，因此head之前的页是最早换入的页
10     if (entry != head) {
11         list_del(entry); //从页链表中删去这一页
12         *ptr_page = le2page(entry, pra_page_link); //ptr_page记录换出的页
13     } else {
14         *ptr_page = NULL;
15     }
16     return 0;
17 }
```

其中有一些可以进一步解释的函数：

1. **list_prev**:该函数较简单，即找到head的前一个链表项，如果删去，会导致该算法不符合FIFO思想，因为在新加入页时我们总是会将其加入head之后，这意味着head的前一项会是最早加入的页；
2. **list_del**:也较为简单，即从该链表中删去某一项
3. **le2page**：这实际上是一个宏定义的函数，是to_struct宏定义函数针对page结构体的进一步封装，作用是找到成员变量pra_page_link=entry的page，即从成员变量找整个结构体，在fifo_out中表示从entry找到将要被换出的页：

```
1 #define le2page(le, member)                 \
2     to_struct((le), struct Page, member)
```

这里的 `head` 我们可以从 `init` 中找到定义，其实就是我们定义的 `pra_list_head` 的链表头

```
1 list_entry_t pra_list_head;
2 /*
3  * (2) _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr
4  *              Now, From the memory control struct mm_struct, we can access FIF
5  */
6 static int
7 _fifo_init_mm(struct mm_struct *mm)
8 {
9      list_init(&pra_list_head);
```

```
10      mm->sm_priv = &pra_list_head;
11      //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
12      return 0;
13  }
```

练习2：深入理解不同分页模式的工作原理（思考题）

正如我们前面所分析的，`get_pte`函数为我们查找并创建页表项，下面我们对其进行进一步的理解与分析，所有的信息我们都写到了注释里面

```
1  pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
2       //pgdir是三级页表的基地址，首先通过它找到第一级页表项
3      pde_t *pdep1 = &pgdir[PDX1(la)];
4      //如果按位与的结果不是有效的，就说明里面存的不是有效的页表项
5      if (!(*pdep1 & PTE_V)) {
6          struct Page *page;
7          //不论什么原因分配失败，都返回空
8          if (!create || (page = alloc_page()) == NULL) {
9              return NULL;
10         }
11
12         //如果分配成功，则需要创建页表项（创建pte【因为pte pde格式类似，可以用这个函数进
13         set_page_ref(page, 1);
14
15         //get the physical address of memory which this (struct* Page *) page  m
16         uintptr_t pa = page2pa(page);
17
18         // sets the first n bytes of the memory area pointed by s
19         memset(KADDR(pa), 0, PGSIZE);
20
21
22         *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
23     }
24
25     //接着通过查找的结果去找二级页表项，操作与上一段完全一样，因为sv39是两级目录
26     pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
27 //    pde_t *pdep0 = &((pde_t *)(PDE_ADDR(*pdep1)))[PDX0(la)];
28     if (!(*pdep0 & PTE_V)) {
29             struct Page *page;
30             if (!create || (page = alloc_page()) == NULL) {
31                     return NULL;
```

```
32              }
33              set_page_ref(page, 1);
34              uintptr_t pa = page2pa(page);
35              memset(KADDR(pa), 0, PGSIZE);
36  //            memset(pa, 0, PGSIZE);
37              *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
38      }
39      return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
40  }
```

解释这两段代码为什么如此相像。

Sv32是用于32位系统，Sv39和Sv48用于64位系统，其中Sv32和sv39均是三级页表，而Sv48是四级页表，在由虚拟地址映射到物理地址过程中，会经历一个递归的寻找过程，由三级页表项找到二级页表，再由二级页表项找到一级页表...因为sv39是三级页表结构，所以会两段大致相同的代码，分别用于通过三级页表项查到二级页表，和由二级页表查到一级页表。

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

将页表项的查找和分配放在一个页表项中挺好的。通过一个控制是否分配的参数的传入决定是否分配页，不仅仅简化了编程的流程，减少了很多判断情况（如查找失败重新分配，此时需要再次查询页表得到对应的位置），方便了代码的管理和调试，也减少了重复的操作。

练习3：给未被映射的地址映射上物理页（需要编程）

见github

PTE和PDE对页面置换的潜在用途

此处，ucore偷懒得将页面应该存放的物理地址存在pte和pde中，只要页表（目录）项中的有效位（V位）为1，那么它们存放的就是物理页有效物理地址，而如果有效位为0，那么它们存放的就是页被换入换出的磁盘地址。

出现了页访问异常，请问硬件要做哪些事情

首先，硬件会检查这个虚拟地址是不是一个合法的地址，以及权限够不够访问这个地址，之后，CPU会通过加载的程序头部分找到这个地址在磁盘中的地址（ucore查找V为0的pte0），此时，硬件找到了空闲页，并将磁盘中的数据移到页中，并与之建立新的页表项映射。

Page的对应关系

一个结构体page会对应一块PGSIZE的物理地址，而该物理地址又由一个PGSIZE大小的虚拟地址空间映射；而该物理地址的页号又会存于一个PTE中，该PTE又可以从一个PDE向下递归找到。

练习4：补充完成Clock页替换算法（需要编程）设计过程

补充函数

在框架中，要求我们补充的是

_clock_init_mm(struct mm_struct *mm)

_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)

_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)

这三个函数。函数的实现过程可以仿照FIFO进行填充，但也有一些不同的地方。

```
1  static int
2  _clock_init_mm(struct mm_struct *mm)
3  {
4      // 初始化pra_list_head为空链表
5      list_init(&pra_list_head);
6      // 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
7      curr_ptr = &pra_list_head;
8      // 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
9      mm->sm_priv = &pra_list_head;
10
11     return 0;
12 }
```

相比于FIFO的填写，Clock算法多出了一个指向当前链表项的指针curr_ptr，指向链表头节点即可，其他的和FIFO的类似。

```
1  static int
2  _clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, in
3  {
4      list_entry_t *head=(list_entry_t*) mm->sm_priv;
5      list_entry_t *entry=&(page->pra_page_link);
6
```

```
    assert(entry != NULL && curr_ptr != 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
    list_add(head, entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    // page->flags = 1;
    page->visited = 1;

    return 0;
}
```

我们按照提示进行实现，这个实现和FIFO进行对比，也仅仅在_clock_map_swappable函数调用时，将page的visited位置为1表示最近被访问。

但依据Clock页替换算法的思想，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为“0”，继续访问下一个页。但我们通过观察_clock_map_swappable的函数调用执行流，该函数并非在每次访问对应页时都执行，这是实验仍然存在问题的地方。

```
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tic
{
     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
    struct Page* p = NULL;
    //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
        for( ; ; curr_ptr = list_prev(curr_ptr)){
            if(curr_ptr == head){
                continue;
            }
            cprintf("curr_ptr %p\n", curr_ptr);
            p = le2page(curr_ptr, pra_page_link);
```

```
            if(p->visited == 1){
                p->visited = 0;
            }else if(p->visited == 0){
                list_del(curr_ptr);
                curr_ptr = list_prev(curr_ptr);
                *ptr_page = p;
                return 0;
            }
        }
        // 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
        // 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
    //}
    *ptr_page = NULL;
    return 0;
}
```

_clock_swap_out_victim函数选择需要被置换的页面。FIFO中我们仅仅选择链表末尾的也进行置换；而依照Clock页替换算法的思想，我们需要在发生页替换时选取最早未被访问的页面，由于需要对访问位进行定期清0，这里按照提示的逻辑：

1. 如果未被访问（即找到的第一个符合要求的页），就从链表中删除该页并将对应的页面指针返回以便后续的操作。

2. 如果页面被访问，就进行清0操作。

进行补全，即完成了对应的算法，并且可以成功make grade。

总结

类似FIFO算法，Clock页替换算法和FIFO算法比较起来实现是相对复杂的，但又是FIFO和LRU的折中。

同时我们需要注意的是，在我们的实验中，也并没有完全实现clock页替换算法，完整的clock页替换算法需要我们在每次访问页面时完成以下两个操作：①要更新对应的visited标志位，②每隔一定的时间间隔后都要重新检测标志位。

这里我们的实验框架，对于②即“定期”这一点进行了取巧但又不失效率的实现，即是在需要进行页替换时遍历页并进行visited位清零变为未被访问的操作。但对于①并没有在每次访问页时都对visited进行修改，没有进行良好的实现，也没有进行框架上的设置，这也是在后续的LRU的实现上出现了一些问题的原因所在。

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？

一个大页的优势：

1. **性能提升：** 采用一个大页可以降低页表的查询次数，分级页表每次要访问多次页表才能获得最后的地址。这可以提高内存访问的效率，减少了页表查找的开销。
2. **内存管理开销小：** 分级页表的管理耗费比较大，一个大页可以减少内核管理页表的开销。

一个大页的缺点：

1. **页表在内存中连续存储，在内存较大时会浪费珍贵的连续内存空间**。一级页表需要分配足够多的连续内存来存储整个虚拟地址空间的映射，这会浪费大量内存，尤其是对于大型地址空间。多级页表实际上是增加了索引，有了索引就可以定位到具体的项。随着虚拟地址空间的增大，存放页表所需要的连续空间也会增大，在操作系统内存紧张或者内存碎片较多时，这无疑会带来额外的开销。但是如果使用多级页表，我们可以使用一页来存放页目录项，页表项存放在内存中的其他位置，不用保证页目录项和页表项连续。
2. **局部性和灵活性较差：**分级页表的话，可以根据实际需求动态分配和销毁各级页表，从而支持稀疏虚拟地址空间，减少内存浪费。分级页表有助于提高局部性，因为相邻的虚拟地址通常映射到相邻的物理地址。

Challenge：实现不考虑实现开销和效率的LRU页替换算法（需要编程）

本次challenge遇到的最困难的问题是：如何在正常访问内存时设置`swap_manager`中相关链表上物理页的LRU？

一种想法是，在find_vma函数中进行相应的修改，在每次寻找虚拟页时维护一个记录本次访问的时间的变量，由此就可以达到目的，但计时的工作这如何进行呢？

我们试图寻找了以下几种方法：

- c库中的time.h等计时方法获取系统当前的时间。该方法在当前的make框架下不能正确导入识别库中定义的变量和函数，将对应的头文件粘贴到/include目录下也无法成功导入。这种方法可能有效。但实际在操作系统的实现中，应该也不会这样去实现。
- lab自带的计时器。由前面的lab也提到了lab自带的clock.c实现的计时器在更晚的时候才进行初始化，显然是不行的。

- 维护一个变量，在一些特定的情况下会执行自动+1来模拟计时。但由于我们还没有实现进程的操作等，这一种想法的可操作性也较差。
- qemu自带的计时器，和C库中计时器一样，不太合理。

上面的几种方法都要求我们维护一个time变量，或者获得一个随时间增加的变量值来实现LRU的逻辑，但最后都没能得以实现。尤其是自己实现一个计时器的想法在没有进程的情况下比较的困难。

另一种想法是，类似clock算法中，我们需要对visited变量进行修改，每次遍历并非该页，就进行+1操作。

这种情况下，代码不能在find_vma中进行编写，原因是find_vma的调用还包括了更早的阶段，如果对find_vma进行大幅度的改动，就会在前期就引起系统的中断。

因此我们认为，只在已有框架下进行编写，是无法完成二次访问已在链表中的页时，对一个可能存储最近一次访问时间的变量进行修改的，而是需要另外定义一些方法。

出于简便，由于lab3中我们还没有设计用户态执行的程序和进程，而是在check_swap 中在内核中分配一些页，模拟对这些页的访问，然后通过 do_pgfault 来调用 swap_map_swappable 函数来查询这些页的访问情况并间接调用相关函数，换出“不常用”的页到磁盘上。所以我们在实验三中也仅仅在check_swap 函数中调用我们新编写的api来模拟对这些页的LRU访问，来体现LRU算法的思想。

我们先给出函数的逻辑，再给出样例测试：

算法逻辑

我们编写的模拟LRU访问的函数为：

```
1  void lru_access(int addr, int val){
2      *(unsigned char *)addr = val;
3      lru_update(addr);
4  }
5
6  void lru_update(int addr){
7      struct Page* pg = get_page(check_mm_struct->pgdir, addr, NULL);
8      if(check_mm_struct!=NULL){
9          list_entry_t *head=(list_entry_t*) check_mm_struct->sm_priv;
10         assert(head != NULL);
11         list_entry_t *le = head->next;
12         // 遍历check_mm_struct的链表，visited加1,若是刚刚访问的addr，visited改为0
13         while (le!=head) {
14             struct Page* page = le2page(le,pra_page_link);
15             if(page!=pg) page->visited++;
16             else {
17                page->visited =0;
18                cprintf("visited ref=%d pra_vaddr=%p\n",page->ref,page->pra_vaddr
19             }
```

```
            le = le->next;
        }
        return;
    }
    return;

}
```

我们在check测试函数中每次调用lru_access函数进行写修改，内部继续调用了lru_update()的更新函数。这个函数对于某个对应地址所归属的页面进行操作，由于visited变量仅仅在此处用到，并没有别的含义，我们可以直接使用visited变量值记为距离最近一次访问该页时，链表被访问的次数，这个变量可以衡量最近访问时间。因此，我们遍历链表，对于并非本次需要访问的页面，我们令其visited变量值+1，否则清零代表刚刚访问。遍历完一遍链表后即说明本次访问的终止。而如何判断是否是我们本次需要访问的页呢？我们可以对获得的Page指针的值进行比较，就可以简易的得到想要的结果，而非对于Page内的一些信息的比较，这样可能会复杂一些。

测试用例及验证

由于check_swap的调用过程中，check过程分为了两部分：

check_content_set()函数和check_content_access()函数，因此我们都需要进行一定的修改。用例如下：

```
static inline void
check_content_set(void)
{
     if(strcmp(sm->name,"lru swap manager")==0){
          lru_access(0x1000, 0x0a);
          assert(pgfault_num==1);
          lru_access(0x1010, 0x0a);
          assert(pgfault_num==1);
          lru_access(0x2000, 0x0b);
          assert(pgfault_num==2);
          lru_access(0x2010, 0x0b);
          assert(pgfault_num==2);
          lru_access(0x3000, 0x0c);
          assert(pgfault_num==3);
          lru_access(0x3010, 0x0c);
```

```
16          assert(pgfault_num==3);
17          lru_access(0x4000, 0x0d);
18          assert(pgfault_num==4);
19          lru_access(0x4010, 0x0d);
20          assert(pgfault_num==4);
21      }
22      else{
23      *(unsigned char *)0x1000 = 0x0a;
24      assert(pgfault_num==1);
25      *(unsigned char *)0x1010 = 0x0a;
26      assert(pgfault_num==1);
27      *(unsigned char *)0x2000 = 0x0b;
28      assert(pgfault_num==2);
29      *(unsigned char *)0x2010 = 0x0b;
30      assert(pgfault_num==2);
31      *(unsigned char *)0x3000 = 0x0c;
32      assert(pgfault_num==3);
33      *(unsigned char *)0x3010 = 0x0c;
34      assert(pgfault_num==3);
35      *(unsigned char *)0x4000 = 0x0d;
36      assert(pgfault_num==4);
37      *(unsigned char *)0x4010 = 0x0d;
38      assert(pgfault_num==4);
39      }
40  }
41
42  // in /kern/mm/swap_lru.c 自行编写的样例。
43  static int
44  _lru_check_swap(void) {
45      cprintf("write Virt Page c in lru_check_swap\n");
46      lru_access(0x3000, 0x0c);
47      assert(pgfault_num==4);
48      cprintf("write Virt Page a in lru_check_swap\n");
49      lru_access(0x1000, 0x0a);
50      assert(pgfault_num==4);
51      cprintf("write Virt Page d in lru_check_swap\n");
52      lru_access(0x4000, 0x0d);
53      assert(pgfault_num==4);
54      cprintf("write Virt Page b in lru_check_swap\n");
55      lru_access(0x2000, 0x0b);
56      assert(pgfault_num==4);
57      cprintf("write Virt Page e in lru_check_swap\n");
58      lru_access(0x5000, 0x0e);
59      assert(pgfault_num==5);
60      cprintf("write Virt Page b in lru_check_swap\n");
61      lru_access(0x3000, 0x0c);
62      assert(pgfault_num==6);
```

```
63      cprintf("write Virt Page a in lru_check_swap\n");
64      lru_access(0x1000, 0x0a);
65      assert(pgfault_num==7);
66      cprintf("write Virt Page b in lru_check_swap\n");
67      lru_access(0x2000, 0x0b);
68      assert(pgfault_num==7);
69      cprintf("write Virt Page c in lru_check_swap\n");
70      lru_access(0x3000, 0x0c);
71      assert(pgfault_num==7);
72      cprintf("write Virt Page d in lru_check_swap\n");
73      lru_access(0x4000, 0x0d);
74      assert(pgfault_num==8);
75      cprintf("write Virt Page e in lru_check_swap\n");
76      lru_access(0x5000, 0x0e);
77      assert(pgfault_num==9);
78      cprintf("write Virt Page a in lru_check_swap\n");
79      assert(*(unsigned char *)0x1000 == 0x0a);
80      lru_access(0x1000, 0x0a);
81      assert(pgfault_num==10);
82      return 0;
83  }
```

对于用例的验证图如下：

```
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: choose victim page 0xc0225908
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
write Virt Page b in lru_check_swap
Store/AMO page fault
page fault at 0x00003000: K/W
SWAP: choose victim page 0xc0225878
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page a in lru_check_swap
Store/AMO page fault
page fault at 0x00001000: K/W
SWAP: choose victim page 0xc0225950
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in lru_check_swap
write Virt Page c in lru_check_swap
write Virt Page d in lru_check_swap
Store/AMO page fault
page fault at 0x00004000: K/W
SWAP: choose victim page 0xc0225908
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: choose victim page 0xc0225950
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: choose victim page 0xc02258c0
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!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
```