Os-lab3

实验

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成 Page Fault 异常处理和部分页面替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。如果大家有余力，可以尝试完成扩展练习，实现 LRU 页替换算法。

实验目的如下：

- 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

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成 Page Fault异常处理和FIFO页替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚 存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操 作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统 系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。

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

```
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

1. `list_init(&pra_list_head)` 初始化了一个名为 `pra_list_head` 的链表。这个链表可能用于在 FIFO 算法中维护进程的内存页面的顺序。

2. `mm->sm_priv = &pra_list_head` 将进程的 `sm_priv` 成员变量设置为指向 `pra_list_head`。这样，进程的内存管理模块可以通过 `sm_priv` 引用到 FIFO 算法所使用的链表来维护页面的顺序。

```
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;
}
```

1.通过 `(list_entry_t*) mm->sm_priv` 将先前在 `_fifo_init_mm` 函数中初始化的链表头指针 `pra_list_head` 赋值给 `head`。这样就能够获取进程的内存管理模块中保存的链表头。

2.`entry` 是形参中传入的 `struct Page` 类型的页面结构体指针 `page` 的 `pra_page_link` 成员。`pra_page_link` 可能是 `struct Page` 中保留的用于链表连接的成员。

3.使用断言 （`assert`） 来确保 `entry` 和 `head` 不为 `NULL`，即确保页面 `page` 的链表成员 `pra_page_link` 和进程的链表头 `head` 存在。

4.在 FIFO 算法中，将最近到达的页面放置在链表末尾。通过调用 `list_add(head, entry)` 将 `entry` 连接到 `head` 链表的末尾。

5.最后，函数返回 0，表示执行成功。

1. `list_init(&pra_list_head)` 初始化了一个名为 `pra_list_head` 的链表。这个链表可能用于在 FIFO 算法中维护进程的内存页面的顺序。

2. `mm->sm_priv = &pra_list_head` 将进程的 `sm_priv` 成员变量设置为指向 `pra_list_head`。这样，进程的内存管理模块可以通过 `sm_priv` 引用到 FIFO 算法所使用的链表来维护页面的顺序。

```
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;
}
```

通过 `(list_entry_t*) mm->sm_priv` 将先前在 `_fifo_init_mm` 函数中初始化的链表头指针 `pra_list_head` 赋值给 `head`。这样就能够获取进程的内存管理模块中保存的链表头。

3. `entry` 是形参中传入的 `struct Page` 类型的页面结构体指针 `page` 的 `pra_page_link` 成员。`pra_page_link` 可能是 `struct Page` 中保留的用于链表连接的成员。
4. 使用断言 （`assert`） 来确保 `entry` 和 `head` 不为 `NULL`，即确保页面 `page` 的链表成员 `pra_page_link` 和进程的链表头 `head` 存在。
5. 在 FIFO 算法中，将最近到达的页面放置在链表末尾。通过调用 `list_add(head, entry)` 将 `entry` 连接到 `head` 链表的末尾。
6. 最后，函数返回 0，表示执行成功。

将某个页面 `page` 添加到先入先出（FIFO）算法维护的链表中的末尾，以记录页面的访问情况。它实现了先入先出策略中的页面替换机制。

```
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;
}
```

这段代码是先入先出（FIFO）算法中的一个函数 `_fifo_swap_out_victim`，根据提供的代码，可以分析出以下内容：

1. 通过 `(list_entry_t*) mm->sm_priv` 将先前在 `_fifo_init_mm` 函数中初始化的链表头指针 `pra_list_head` 赋值给 `head`。
2. 使用断言 `assert` 来确保 `head` 不为 `NULL`，即确保链表头 `head` 存在。
3. 使用断言 `assert` 来确保 `in_tick` 的值为 0。

4. 在 FIFO 算法中，选择替换的页面时，需要选择最早到达的页面（链表头的下一个页面）。通过调用 `list_prev(head)` 获取链表头 `head` 的前一个节点 `entry`。

5. 如果 `entry` 不等于 `head`，表示存在要替换的页面。则执行以下操作：

    - 使用 `list_del(entry)` 将 `entry` 从链表中删除。
    - 使用 `le2page(entry, pra_page_link)` 获取被删除页面的地址，并将其赋值给 `*ptr_page`。`le2page` 是一个将链表节点转换为 `struct Page` 指针的宏定义。

6. 如果 `entry` 等于 `head`，表示链表中没有页面可供替换，则将 `*ptr_page` 置为 `NULL`。

7. 最后，函数返回 0，表示执行成功。

总体而言，这段代码的作用是在先入先出（FIFO）算法中选择一个最早到达的页面作为替换的受害者，将其从链表中删除，并将该页面的地址通过 `ptr_page` 返回。它实现了先入先出策略中的页面替换机制。需要注意的是，这里只实现了页面的选择和删除，并没有实际进行页面替换的操作。

```
_fifo_init(void)
{
    return 0;
}

static int
_fifo_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}

static int
_fifo_tick_event(struct mm_struct *mm)
{ return 0; }
```

1. `_fifo_init` 函数：这个函数在初始化时被调用，但在提供的代码中函数体为空。它返回 0，表示初始化成功。具体而言，这个函数似乎没有实际操作，可能是为了与其他页面替换算法的函数保持一致而存在。

2. `_fifo_set_unswappable` 函数：这个函数用于设置指定地址对应的页面为不可置换。根据提供的代码，函数体为空，并且它也返回 0，表示设置成功。与 `_fifo_init` 类似，这个函数可能是为了与其他页面替换算法的函数保持一致而存在，并没有实际操作。

3. `_fifo_tick_event` 函数：这个函数在调用时代表一个时钟滴答事件。根据提供的代码，函数体为空，同时返回 0，表示事件处理成功。与前面两个函数类似，这个函数可能是用来与其他页面替换算法的函数保持接口一致性，而实际操作可能在其他地方完成。

```
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
    }
    return 0;
}
```

这段代码是关于页面置换算法中的时钟页面置换算法（Clock Page Replacement Algorithm）的实现函数 `_clock_map_swappable`。

代码分析如下：

1. 首先，将页面的链接指针 `entry` 初始化为指向 `page` 结构中的 `pra_page_link` 成员。
2. 断言（assert）确保 `entry` 和 `curr_ptr`（在代码中未给出）不为 NULL。这里的断言可能用于检查代码的正确性，确保 `entry` 和 `curr_ptr` 在使用前已经被正确初始化。
3. 代码中的注释提示了要实现的功能：记录页面的访问情况，并将最近访问的页面插入到 `pra_list_head` 页面链表的末尾。同时，将页面的 `visited` 标志置为 1，表示该页面已被访问。
4. 最后，函数返回 0，表示函数执行成功。

根据提供的代码，可以看出这个函数实现了时钟页面置换算法的关键步骤，即根据页面的访问情况更新页面链表，并标记页面的访问状态。具体的操作可能需要在其他地方完成，例如更新 `pra_list_head` 页面链表，更新页面的访问标志等。

```
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;
}
```

这段代码是关于交换空间初始化的函数 `swap_init`。

代码分析如下：

1. 调用 `swapfs_init()` 函数进行交换文件系统的初始化。此函数可能负责设置和管理交换文件。
2. 进行一个条件判断，判断最大交换偏移量 `max_swap_offset` 的取值是否符合要求。
    - 如果不符合要求（即不满足 `7 <= max_swap_offset < MAX_SWAP_OFFSET_LIMIT`），触发 panic 错误，打印错误信息。
3. 将 `swap_manager_clock` 赋值给 `sm` 指针，这里可能是选择了时钟页面置换算法作为交换空间的管理算法。
4. 调用 `sm->init()` 方法对选定的交换空间管理算法进行初始化，并将返回值存储在变量 `r` 中。
5. 进行 `r` 的判断。
    - 如果 `r` 的值为 0，表示交换空间管理算法初始化成功。
        - 将 `swap_init_ok` 设置为 1，表示交换空间初始化成功。
        - 打印交换空间管理算法的名称，使用 `cprintf` 函数输出。
        - 调用 `check_swap()` 函数进行交换空间的检查。
6. 返回变量 `r`，表示交换空间初始化的结果。

根据提供的代码片段分析，这段代码的功能是进行交换空间的初始化。首先，初始化交换文件系统；然后判断最大交换偏移量的取值是否符合要求；选择时钟页面置换算法作为交换空间的管理算法，并进行初始化；最后，判断交换空间管理算法初始化的结果，如果成功则设置相应的变量值，并进行交换空间的检查。

```
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;
}
```

这段代码是关于页面置换的函数 `swap_out`。

代码分析如下：

1. 定义变量 `i` 作为循环计数器，初始值为 0。
2. 进行一个循环，循环次数为参数 `n` 的值。
    - 在每次循环开始时定义变量 `v` 存储虚拟地址。
    - 定义变量 `page` 存储要被置换出的页面。
    - 调用 `sm->swap_out_victim` 方法选择要被置换出的页面，并将结果存储在变量 `r` 中。
    - 如果选择页面失败（`r` 不等于 0），输出错误信息并跳出循环。
    - 获取虚拟地址对应的页表项 `ptep`。
    - 判断页表项是否有效。
    - 调用 `swapfs_write` 方法将页面写入磁盘交换空间。
        - 如果写入失败，输出失败信息，然后将页面重新标记为可交换状态，并进行下一次循环。
        - 如果写入成功，输出成功信息，将页表项设置为指向磁盘交换空间的位置，然后释放页面。

- 使用 `tlb_invalidate` 方法使页表项失效。

3. 返回变量 `i` 值，表示成功置换出的页面数量。

根据提供的代码片段分析，这段代码的功能是进行页面置换操作。循环 `n` 次，每次选择一个页面进行置换，将选中的页面写入磁盘交换空间，然后更新页表项，最后使页表项失效。在每次循环中，如果选择页面或写入磁盘交换空间失败，会输出相应的错误信息并继续进行下一次循环。

```
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;
}
```

这段代码是关于页面置换的函数 `swap_in`。

代码分析如下：

1. 定义一个变量 `result` 用来存储分配的页面。
2. 断言 `result` 不为 NULL，确保成功分配页面。
3. 获取虚拟地址 `addr` 对应的页表项 `ptep`。
4. 如果调用 `swapfs_read` 方法读取磁盘交换空间中的页面失败（返回值不等于 0），则断言失败。
5. 输出成功加载磁盘交换空间中的页面的信息。
6. 将 `result` 的值赋给指针变量 `ptr_result`。
7. 返回 0，表示成功进行页面加载。

根据提供的代码片段分析，这段代码的功能是将页面从磁盘交换空间加载到内存中。首先通过 `alloc_page` 分配一个页面，然后获取虚拟地址对应的页表项。接着调用 `swapfs_read` 方法读取磁盘交换空间中的页面，并将读取结果存储在分配的页面中。最后，输出成功加载页面的信息，并将加载的页面通过 `ptr_result` 返回。

```
static void
check_swap(void)
{
    //backup mem env
     int ret, count = 0, total = 0, i;
     list_entry_t *le = &free_list;
     while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        assert(PageProperty(p));
        count ++, total += p->property;
     }
     assert(total == nr_free_pages());
     cprintf("BEGIN check_swap: count %d, total %d\n",count,total);

     //now we set the phy pages env
     struct mm_struct *mm = mm_create();
     assert(mm != NULL);

     extern struct mm_struct *check_mm_struct;
     assert(check_mm_struct == NULL);

     check_mm_struct = mm;

     pde_t *pgdir = mm->pgdir = boot_pgdir;
     assert(pgdir[0] == 0);

     struct vma_struct *vma = vma_create(BEING_CHECK_VALID_VADDR,
CHECK_VALID_VADDR, VM_WRITE | VM_READ);
     assert(vma != NULL);

     insert_vma_struct(mm, vma);

     //setup the temp Page Table vaddr 0~4MB
     cprintf("setup Page Table for vaddr 0X1000, so alloc a page\n");
     pte_t *temp_ptep=NULL;
     temp_ptep = get_pte(mm->pgdir, BEING_CHECK_VALID_VADDR, 1);
     assert(temp_ptep!= NULL);
     cprintf("setup Page Table vaddr 0~4MB OVER!\n");

     for (i=0;i<CHECK_VALID_PHY_PAGE_NUM;i++) {
          check_rp[i] = alloc_page();
          assert(check_rp[i] != NULL );
          assert(!PageProperty(check_rp[i]));
     }
     list_entry_t free_list_store = free_list;
     list_init(&free_list);
     assert(list_empty(&free_list));

     //assert(alloc_page() == NULL);

     unsigned int nr_free_store = nr_free;
     nr_free = 0;
     for (i=0;i<CHECK_VALID_PHY_PAGE_NUM;i++) {
        free_pages(check_rp[i],1);
     }
     assert(nr_free==CHECK_VALID_PHY_PAGE_NUM);
```

```
     cprintf("set up init env for check_swap begin!\n");
     //setup initial vir_page<->phy_page environment for page relpacement
algorithm


     pgfault_num=0;

     check_content_set();
     assert( nr_free == 0);
     for(i = 0; i<MAX_SEQ_NO ; i++)
         swap_out_seq_no[i]=swap_in_seq_no[i]=-1;

     for (i= 0;i<CHECK_VALID_PHY_PAGE_NUM;i++) {
         check_ptep[i]=0;
         check_ptep[i] = get_pte(pgdir, (i+1)*0x1000, 0);
         //cprintf("i %d, check_ptep addr %x, value %x\n", i, check_ptep[i],
*check_ptep[i]);
         assert(check_ptep[i] != NULL);
         assert(pte2page(*check_ptep[i]) == check_rp[i]);
         assert((*check_ptep[i] & PTE_V));
     }
     cprintf("set up init env for check_swap over!\n");
     // now access the virt pages to test  page relpacement algorithm
     ret=check_content_access();
     assert(ret==0);

     //restore kernel mem env
     for (i=0;i<CHECK_VALID_PHY_PAGE_NUM;i++) {
         free_pages(check_rp[i],1);
     }

     //free_page(pte2page(*temp_ptep));

     mm_destroy(mm);

     nr_free = nr_free_store;
     free_list = free_list_store;


     le = &free_list;
     while ((le = list_next(le)) != &free_list) {
         struct Page *p = le2page(le, page_link);
         count --, total -= p->property;
     }
     cprintf("count is %d, total is %d\n",count,total);
     //assert(count == 0);

     cprintf("check_swap() succeeded!\n");
}
```

这段代码是一个用于检查页面置换算法的函数 `check_swap`。

代码分析如下：

1. 备份当前的内存环境：
    - 遍历空闲链表 `free_list`，计算页面数量和总的属性值。
    - 断言总的属性值与可用的页面数量相等。
    - 输出页面数量和总的属性值。
2. 设置物理页面环境：
    - 创建一个新的内存管理结构 `mm`。
    - 断言 `mm` 不为空。
    - 将 `mm` 赋值给全局变量 `check_mm_struct`。
    - 将初始页目录表 `boot_pgdir` 赋值给 `mm` 的页表目录 `pgdir`。
    - 断言 `pgdir[0]` 的值为 0。
    - 创建一个虚拟内存区域 `vma`。
    - 断言 `vma` 不为空。
    - 将 `vma` 插入到 `mm` 的虚拟内存区域链表中。
3. 设置临时页表：
    - 分配一个页面作为临时页表。
    - 获取虚拟地址 `BEING_CHECK_VALID_VADDR` 对应的页表项指针 `temp_ptep`。
    - 断言 `temp_ptep` 不为空。
4. 分配一定数量的物理页面：
    - 使用 `alloc_page` 分配一定数量的页面，将页面指针存储在 `check_rp` 数组中。
    - 断言页面指针不为空且页面不具有属性。
5. 保存当前的空闲链表，并清空空闲链表：
    - 创建一个临时变量 `free_list_store` 存储当前的空闲链表。
    - 初始化空闲链表为空。
6. 修改空闲页面数量：
    - 保存当前的空闲页面数量到临时变量 `nr_free_store`。
    - 将空闲页面数量设为 0。
7. 释放之前分配的物理页面：
    - 使用 `free_pages` 释放之前分配的页面。
8. 设置初始虚拟页面和物理页面的映射关系：
    - 调用 `check_content_set` 函数设置初始的虚拟页面和物理页面的映射关系。
    - 断言当前的空闲页面数量为 0。
9. 初始化交换序列号数组：
    - 将交换序列号数组 `swap_out_seq_no` 和 `swap_in_seq_no` 的元素都设为 -1。
10. 获取并检查临时页表项：

- 遍历物理页面数组 `check_rp`，依次获取相应的页表项指针 `check_ptep[i]`。
- 断言页表项指针不为空。
- 断言页表项指针指向的物理页与之前分配的页面相等。
- 断言页表项标志位含有 PTE_V（有效位）标志。

11. 访问虚拟页面，测试页面置换算法：

- 调用 `check_content_access` 函数访问虚拟页面，测试页面置换算法的效果。
- 断言返回值为 0，表示访问成功。

12. 恢复内核内存环境：

- 释放之前分配的物理页面。
- 销毁内存管理结构 `mm`。
- 还原空闲页面数量和空闲链表。

13. 检查页面属性计算的正确性：

- 遍历空闲链表，计算页面数量和总的属性值。
- 断言页面数量和总的属性值相符。
- 输出计算的页面数量和总的属性值。

14. 输出检查成功的信息。

根据提供的代码片段分析，这段代码的功能是对页面置换算法进行检查和测试。函数首先备份当前的内存环境，然后设置物理页面环境和临时页表。接着分配一定数量的物理页面，并清空空闲链表。然后设置初始的虚拟页面和物理页面的映射关系，并进行页面访问以测试页面置换算法。最后恢复内核内存环境并检查页面属性计算的正确性。

```
static void page_init(void) {
    extern char kern_entry[];

    va_pa_offset = KERNBASE - 0x80200000;
    uint64_t mem_begin = KERNEL_BEGIN_PADDR;
    uint64_t mem_size = PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR;
    uint64_t mem_end = PHYSICAL_MEMORY_END; //硬编码取代 sbi_query_memory()接口
    cprintf("membegin %llx memend %llx mem_size %llx\n",mem_begin, mem_end,
mem_size);
    cprintf("physcial memory map:\n");
    cprintf("  memory: 0x%08lx, [0x%08lx, 0x%08lx].\n", mem_size, mem_begin,
            mem_end - 1);
    uint64_t maxpa = mem_end;

    if (maxpa > KERNTOP) {
        maxpa = KERNTOP;
    }

    extern char end[];

    npage = maxpa / PGSIZE;
    // BBL has put the initial page table at the first available page after
the
    // kernel
```

```
    // so stay away from it by adding extra offset to end
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
    for (size_t i = 0; i < npage - nbase; i++) {
        SetPageReserved(pages + i);
    }

    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * (npage
- nbase));
    mem_begin = ROUNDUP(freemem, PGSIZE);
    mem_end = ROUNDDOWN(mem_end, PGSIZE);
    if (freemem < mem_end) {
        init_memmap(pa2page(mem_begin), (mem_end - mem_begin) / PGSIZE);
    }
}
```

这段代码是用于初始化页面管理的函数 `page_init`。

代码分析如下：

1. 外部符号与变量：
    - 引用外部符号 `kern_entry`。
2. 计算地址偏移量：
    - 计算虚拟地址和物理地址之间的偏移量，将偏移量存储在全局变量 `va_pa_offset` 中。
    - 偏移量的计算公式为 `KERNBASE - 0x80200000`。
3. 设置内存参数：
    - 定义变量 `mem_begin`，存储内核开始的物理地址，值为 `KERNEL_BEGIN_PADDR`。
    - 定义变量 `mem_size`，存储内核占用的物理内存大小，值为 `PHYSICAL_MEMORY_END - KERNEL_BEGIN_PADDR`。
    - 定义变量 `mem_end`，存储整个物理内存的结束地址，值为 `PHYSICAL_MEMORY_END`。
    - 输出内存相关信息。
4. 确定最大的物理地址：
    - 将 `maxpa` 初始化为 `mem_end`。
    - 如果 `maxpa` 大于 `KERNTOP`，则将其设为 `KERNTOP`。
    - `KERNTOP` 是内核代码结束的地址。
5. 计算总的页面数量：
    - 根据 `maxpa` 的大小，计算页面数量 `npage`。即 `maxpa / PGSIZE`。
    - `PGSIZE` 是页面的大小。
6. 分配页面数据结构的内存：
    - 定义指针变量 `pages`，指向页面数据结构的起始位置。
    - 根据 `end`（内核代码结束后的地址）向上对齐，并将其作为页面数据结构的起始地址。
    - 遍历从 `pages` 开始的页面数据结构，设置这些页面为保留页面（reserved）。

7. 计算可用内存的起始地址：

   - 计算 `freemem`，指向页面数据结构之后可供使用的内存起始地址。
   - 约束 `freemem` 需要对齐至页面边界。

8. 初始化内存映射：

   - 如果可用内存起始地址 `freemem` 小于整个物理内存的结束地址 `mem_end`，则调用 `init_memmap` 函数初始化内存映射。
   - `init_memmap` 函数用于将一段物理内存划分为多个页面，并根据需要设置页面属性。

通过分析，这段代码的功能是根据内核代码的位置和整个物理内存空间的大小，初始化页面管理相关的数据结构和内存映射。其中包括计算地址偏移量、确定可用的物理内存范围、分配页面数据结构的内存、初始化保留页面，并根据剩余可用内存初始化内存映射。

```
static void boot_map_segment(pde_t *pgdir, uintptr_t la, size_t size,
                             uintptr_t pa, uint32_t perm) {
    assert(PGOFF(la) == PGOFF(pa));
    size_t n = ROUNDUP(size + PGOFF(la), PGSIZE) / PGSIZE;
    la = ROUNDDOWN(la, PGSIZE);
    pa = ROUNDDOWN(pa, PGSIZE);
    for (; n > 0; n--, la += PGSIZE, pa += PGSIZE) {
        pte_t *ptep = get_pte(pgdir, la, 1);
        assert(ptep != NULL);
        *ptep = pte_create(pa >> PGSHIFT, PTE_V | perm);
    }
}
```

这段代码是用于在页表中映射一个段（segment）的函数 `boot_map_segment`。

代码分析如下：

1. 函数参数：

   - `pgdir`：页表的起始地址，`pde_t` 类型的指针。
   - `la`：线性地址（虚拟地址）。
   - `size`：段的大小。
   - `pa`：物理地址。
   - `perm`：控制页面权限的位掩码。

2. 断言检查偏移：

   - 使用断言检查 `la` 和 `pa` 的页内偏移是否相同，即 `PGOFF(la) == PGOFF(pa)`。
   - `PGOFF(addr)` 是获得 `addr` 的页内偏移。

3. 计算页面数量和对齐地址：

   - 计算需要映射的页面数量 `n`，将段的大小和页内偏移取整到页面大小（`PGSIZE`）的倍数，然后除以 `PGSIZE`。
   - 更新 `la` 和 `pa` 的值，将它们向下取整到页面的起始地址，使用 `ROUNDDOWN(addr, PGSIZE)` 来实现。

4. 遍历映射页面：

    - 使用一个循环，迭代从 `n` 到 1。
    - 在每次循环中，递增 `la` 和 `pa` 的值，以页面大小为步长。
    - 调用 `get_pte` 函数获取 `la` 所对应的页表项的地址，如果页表项不存在，则会创建一个新的页表项。
    - 使用 `pte_create` 函数创建一个新的页表项，并将其值存储在获取到的页表项地址上。
    - `pte_create` 函数用于根据给定的物理页帧号和权限位掩码创建一个页表项。

通过分析，这段代码的功能是将一个大小为 `size` 的段从虚拟地址 `la` 映射到物理地址 `pa`，并给映射的页面设置权限 `perm`。它遍历需要映射的页面，并在页表中创建相应的页表项，将虚拟地址和物理地址进行映射。

```
static void *boot_alloc_page(void) {
    struct Page *p = alloc_page();
    if (p == NULL) {
        panic("boot_alloc_page failed.\n");
    }
    return page2kva(p);
}
```

这段代码是用于在引导阶段（boot）分配一个页面的函数 `boot_alloc_page`。

代码分析如下：

1. 分配页面：

    - 调用函数 `alloc_page()` 来分配一个页面，返回一个指向 `struct Page` 结构体的指针，表示分配的物理页面。
    - 将返回的指针保存在变量 `p` 中。

2. 检查分配是否成功：

    - 判断变量 `p` 是否为 `NULL`，即分配是否失败。
    - 如果分配失败，则触发一个恐慌（panic），输出错误信息 "boot_alloc_page failed.\n"。

3. 返回虚拟地址：

    - 调用函数 `page2kva()`，将 `struct Page` 结构体指针 `p` 转换为与之对应的虚拟地址。
    - 返回转换后的虚拟地址。

通过分析，这段代码的功能是在引导阶段分配一个物理页面，并返回与该页面对应的虚拟地址。它通过调用 `alloc_page()` 函数来分配页面，并使用 `page2kva()` 函数将物理地址转换为虚拟地址。

```
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)];
        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);
        }
        return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
    }
```

这段代码是用于获取页面表项(`pte_t`)的函数`get_pte`。以下是对代码逐行的分析：

1. 获取一级页表项指针(`pdep1`)：
    - 使用宏定义 `PDX1(la)` 将虚拟地址 `la` 转换为一级页表项的索引。
    - 通过 `pgdir` 参数获取 `pdep1`，即一级页表项的地址。
2. 检查一级页表项是否有效：
    - 通过按位与运算 `*pdep1 & PTE_V` 判断一级页表项是否有效（存在）。
    - 如果一级页表项无效，说明对应的页表不存在。
3. 分配一级页表项：
    - 声明一个指向 `struct Page` 结构体的指针 `page`。
    - 如果 `create` 参数为 `false`，或者分配失败（`alloc_page()` 返回 `NULL`），则返回 `NULL`。
    - 如果分配成功，将页面的引用计数设置为 1（`set_page_ref(page, 1)`）。
    - 获取页面的物理地址 `pa`（`uintptr_t pa = page2pa(page)`）。
    - 使用 `memset()` 函数将页面内容清零（`memset(KADDR(pa), 0, PGSIZE)`）。
    - 使用 `pte_create()` 宏定义创建一个二级页表项的值，包括页号和权限标志（`PTE_U | PTE_V`），并赋值给对应的一级页表项 `*pdep1`。
4. 获取二级页表项指针(`pdep0`)：
    - 使用宏定义 `PDX0(la)` 将虚拟地址 `la` 转换为二级页表项的索引。
    - 通过 `KADDR()` 宏定义将一级页表项中保存的物理地址转换为内核虚拟地址，然后根据索引获取二级页表项的地址。
5. 检查二级页表项是否有效：
    - 通过按位与运算 `*pdep0 & PTE_V` 判断二级页表项是否有效（存在）。
    - 如果二级页表项无效，说明对应的页表不存在。

6. 分配二级页表项：

   - 这部分代码与分配一级页表项的步骤类似，具体的操作流程和实现细节与步骤 3 相同。

7. 返回页面表项指针：

   - 使用宏定义 `PTX(la)` 将虚拟地址 `la` 转换为页面表项的索引。
   - 通过 `KADDR()` 宏定义将二级页表项中保存的物理地址转换为内核虚拟地址，然后根据索引获取页面表项的地址，并返回。

```
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
    }
}
```

这段代码是用来删除页面表项（pte_t）的函数`page_remove_pte`。以下是对代码逐行的分析：

1. 检查页面表项是否有效：

   - 通过按位与运算 `*ptep & PTE_V` 检查页面表项是否有效（存在）。
   - 如果页面表项无效，则表示对应的页表不存在，不需要执行任何操作。

2. 获取对应的页面结构指针：

   - 使用宏定义 `pte2page()` 将页面表项的值转换为对应的页面结构指针 `page`。
   - 这里的 `pte2page()` 宏定义用于根据页面表项的值获取对应的页面结构指针。

3. 减少页面的引用计数：

   - 使用 `page_ref_dec()` 函数将页面的引用计数减少。
   - 当页面的引用计数减少到0时，表示该页面没有被其他地方引用，可以释放。

4. 检查页面引用计数：

   - 使用 `page_ref()` 函数获取页面的引用计数。
   - 如果页面的引用计数为0，说明没有其他地方再引用该页面。

5. 清除页面表项：

   - 将页面表项 `*ptep` 的值设为0，即清除该页面表项的内容。

6. 刷新TLB缓存：

- 使用 `tlb_invalidate()` 函数刷新TLB缓存，确保之前缓存的虚拟地址到物理地址的映射无效。
- TLB缓存是用于加速虚拟地址到物理地址的转换，当页面表项被修改或移除时，需要手动刷新TLB缓存，以保证后续访问正确的映射。

总之，这段代码的作用是删除指定页面表项，并处理相应的页面引用计数和TLB缓存。

```
struct Page *pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page();
    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);
        }
    }
    return page;
}
```

这段代码是用于在页表中为给定的线性地址（`la`）分配一个物理页面（`struct Page`），并将其插入到页表中的函数`pgdir_alloc_page`。以下是对代码逐行的分析：

1. 分配一个物理页面：
    - 使用`alloc_page()`函数分配一个物理页面，返回一个指向`struct Page`结构的指针赋值给`page`。
2. 检查页面是否成功分配：
    - 如果分配成功，则继续执行下面的操作；如果分配失败（`page`为`NULL`），则直接返回`NULL`。
3. 将页面插入到页表中：
    - 使用`page_insert()`函数将分配的页面`page`插入到给定的页表`pgdir`中，映射到线性地址`la`，并设置指定的权限`perm`。
    - 如果插入失败（`page_insert()`返回非零值），则释放之前分配的页面`page`，然后返回`NULL`。
4. 检查是否具备进行交换操作的条件：
    - 检查是否已成功初始化交换机制，通过判断变量`swap_init_ok`的值。
    - 如果具备交换操作的条件，则执行以下操作；否则，跳过此部分。
5. 将页面标记为可交换的：
    - 使用`swap_map_swappable()`函数将页面`page`标记为可交换的，与当前进程的内存管理结构`check_mm_struct`相关联，并指定线性地址`la`与页面`page`之间的映射。
    - 设置页面`page`的`pra_vaddr`字段为线性地址`la`，表示页面映射到的线性地址。

- 使用断言（assert()）检查页面的引用计数是否为1，即该页面当前只被一个地方引用。

6. 返回分配的页面指针：

- 无论是否具备交换操作的条件，都会返回之前分配的页面指针page。

总之，这段代码的作用是在给定的页表中为指定的线性地址分配一个物理页面，并将页面插入到页表中进行映射。如果具备交换操作的条件，则还将页面标记为可交换，并进行一些额外的检查和设置。最后，返回分配的页面指针。

```
struct mm_struct *
mm_create(void) {
    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;

        if (swap_init_ok) swap_init_mm(mm);
        else mm->sm_priv = NULL;
    }
    return mm;
}
```

这段代码定义了一个名为mm_create的函数，用于创建一个struct mm_struct类型的内存管理结构体。以下是对代码逐行的分析：

1. 分配内存管理结构体：

- 使用kmalloc()函数分配一块大小为sizeof(struct mm_struct)的内存，并将返回的指针赋值给mm。

2. 检查内存管理结构体是否成功分配：

- 如果成功分配（mm不为NULL），则继续执行下面的操作；如果分配失败，则直接返回NULL。

3. 初始化内存管理结构体的字段：

- 使用list_init()函数初始化mm中的mmap_list，将其设为一个空链表。
- 将mm的mmap_cache字段设为NULL，表示缓存为空。
- 将mm的pgdir字段设为NULL，表示页表尚未创建。
- 将mm的map_count字段设为0，表示内存映射计数为0。

4. 检查是否具备进行交换操作的条件：

- 检查是否已成功初始化交换机制，通过判断变量swap_init_ok的值。
- 如果具备交换操作的条件，则调用swap_init_mm()函数初始化mm的交换机制私有数据（sm_priv）；否则，将mm的sm_priv字段设为NULL。

5. 返回内存管理结构体指针：

- 无论是否具备交换操作的条件，都会返回创建的内存管理结构体指针`mm`。

总之，这段代码的作用是创建一个`struct mm_struct`类型的内存管理结构体，并进行初始化。其中包括初始化链表、缓存、页表和内存映射计数等字段。如果具备交换操作的条件，则还会初始化交换机制的私有数据。最后，返回创建的内存管理结构体指针。

```
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;
}
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;
}
```

这段代码由两个函数组成：`vma_create`和`find_vma`。我将逐个分析这两个函数的功能：

1. `vma_create`函数：
    - 这个函数用于创建一个`struct vma_struct`类型的虚拟内存区域（VMA）结构体，并进行初始化。
    - 首先，使用`kmalloc()`函数分配一块大小为`sizeof(struct vma_struct)`的内存，并将返回的指针赋值给`vma`。
    - 接下来，检查内存分配是否成功。如果分配成功（`vma`不为`NULL`），则将`vm_start`、`vm_end`和`vm_flags`分别赋值给`vma`的相应字段。

- 最后，返回创建的VMA结构体指针`vma`。

2. `find_vma`函数：

- 这个函数用于在给定的内存管理结构体`mm`中查找与给定地址`addr`匹配的VMA。
- 首先，检查传入的`mm`是否为空。如果为空，直接返回`NULL`。
- 如果`mm`的缓存`mmap_cache`不为空且`addr`位于缓存的VMA范围内（`vma->vm_start <= addr`且`vma->vm_end > addr`），则可以直接返回缓存的VMA。
- 否则，需要遍历整个VMA链表`mmap_list`来查找匹配的VMA。
  - 初始化一个布尔变量`found`为`0`，表示未找到匹配的VMA。
  - 使用循环遍历VMA链表，直到遍历完链表或找到匹配的VMA为止。
  - 对于每个VMA，检查其起始地址和结束地址是否满足条件`vma->vm_start <= addr`且`addr < vma->vm_end`。
  - 如果找到匹配的VMA，将`found`设为`1`，中断循环。
- 如果未找到匹配的VMA，将`vma`设为`NULL`。
- 最后，如果找到了匹配的VMA（`vma`不为空），将其设置为内存管理结构体的缓存`mmap_cache`。
- 返回找到的VMA结构体指针`vma`。

综上所述，这两个函数分别用于创建和查找虚拟内存区域（VMA），并提供了方便的操作接口。

```
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;
        }
    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 ++;
}
```

这段代码实现了将一个VMA插入到内存管理结构体的VMA链表中的功能。我将逐行分析代码的执行过程：

1. 首先，代码使用`assert`断言来确保`vma->vm_start`小于`vma->vm_end`，即VMA的起始地址应该小于结束地址。

2. 接下来，定义了两个指针变量list和le_prev，并将它们都指向内存管理结构体mm的VMA链表。
3. 进入循环，使用指针变量le进行遍历，条件为le的下一个节点不是链表的头节点（即遍历完整个链表）。
4. 在循环中，首先将当前节点转换为VMA结构体指针，命名为mmap_prev。
5. 检查mmap_prev的起始地址是否大于待插入VMA的起始地址。如果是，跳出循环，表示找到了插入位置。
6. 否则，更新le_prev为当前节点，并继续下一次循环。
7. 循环结束后，得到le_prev节点的下一个节点的指针，命名为le_next。
8. 接下来，检查le_prev节点和le_next节点是否存在，即待插入的VMA需要与其前后的VMA进行重叠检查。
9. 如果le_prev不是链表的头节点，调用check_vma_overlap函数来检查待插入VMA与le_prev节点的重叠情况。
10. 如果le_next不是链表的头节点，调用check_vma_overlap函数来检查待插入VMA与le_next节点的重叠情况。
11. 将内存管理结构体mm指针赋值给待插入的VMA的vm_mm字段。
12. 使用list_add_after函数将待插入的VMA插入到le_prev节点之后。
13. 最后，将内存管理结构体mm的map_count字段加一，表示VMA的数量增加了。

综上所述，这段代码的作用是将一个VMA插入到内存管理结构体的VMA链表中，并进行适当的重叠检查。

```
void
mm_destroy(struct mm_struct *mm) {

    list_entry_t *list = &(mm->mmap_list), *le;
    while ((le = list_next(list)) != list) {
        list_del(le);
        kfree(le2vma(le, list_link),sizeof(struct vma_struct));  //kfree vma
    }
    kfree(mm, sizeof(struct mm_struct)); //kfree mm
    mm=NULL;
}
```

这段代码实现了销毁内存管理结构体（struct mm_struct）的功能。让我逐行分析代码的执行过程：

1. 首先，定义了一个指针变量list，并将其指向内存管理结构体mm的VMA链表。
2. 进入循环，条件为le的下一个节点不是链表的头节点（即遍历完整个链表）。
3. 在循环中，首先获取当前节点的指针le，然后使用list_del函数将该节点从链表中移除。

4. 接下来，使用`le2vma`宏将节点指针转换为VMA结构体指针，并使用`kfree`函数释放该VMA的内存空间，同时指定要释放的字节数。

5. 循环结束后，使用`kfree`函数释放内存管理结构体`mm`的内存空间，同样指定要释放的字节数。

综上所述，这段代码的作用是销毁内存管理结构体`mm`，包括释放所有VMA的内存空间，并最终释放`mm`本身的内存空间。

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。

相似代码如下：

```
pde_t *pdep1 = &pgdir[PDX1(la)];
if (!(*pdep1 & PTE_V)) {
    // Allocate a page for the second level page table (if create is true)
}
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
if (!(*pdep0 & PTE_V)) {
    // Allocate a page for the third level page table (if create is true)
}
return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
```

这段代码的目的是获取线性地址 la 对应的页表项的地址。代码首先计算了对应的一级页表项（PDE1）的地址，并检查其是否有效（通过 PTE_V 标志）。如果一级页表项无效，说明对应的二级页表不存在，需要为其分配一个页面。

接着，代码计算了二级页表项（PDE0）的地址，同样检查其是否有效。如果二级页表项无效，说明对应的三级页表不存在，需要为其分配一个页面。

最后，代码通过计算得到的页表项地址，返回对应的页表项指针。无论是在 sv32、sv39 还是 sv48 的情况下，页表的基本结构是类似的。每个页表级别（一级页表和二级页表）都由相应的目录项（PDE）和页表项（PTE）组成。这意味着相似的操作逻辑可以用于不同的页表级别。

而这两段代码的主要区别在于， 第一段代码的开始部分为

```
pde_t *pdep1 = &pgdir[PDX1(la)];
```

第二段代码的开始部分为

```
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
```

这两句代码都是用于获取第一级页目录项（PDE）的地址，但第二段它在前面加了一些转换操作KADDR(PDE_ADDR(...))，此函数在 sv32 中用于将物理地址转换为内核虚拟地址。而在 sv39 和 sv48 中，地址转换会更复杂，因为涉及到了更多的层次和表项。

• 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？

查找PTE和分配PTE是密切相关的操作。所以将他们合并在一起会有以下好处：

1. 保持逻辑一致性 将它们合并在一个函数中可以保持逻辑一致性，使得代码更容易理解和维护。
2. 简化接口调用，减少函数调用开销 将两者合并可以简化对页表的操作，用户只需要调用一个函数就能完成查找或分配页表项的操作。这样做可以减少用户调用的复杂度
3. 减少代码的冗余 查找和分配页表项的内部实现可能涉及到相似的逻辑和数据结构，合并在一起可以更好地共享内部实现，减少代码的冗余。

但是在一些复杂的操作系统或需要更灵活的内存管理场景下，还是要将查找和分配分成两个独立的函数，以便更好地控制内存管理的细节。这取决于具体的实践需求。

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。

简述设计实现过程:

```
    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;
            //(1) According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            //(3) make the page swappable.
            swap_in(mm, addr, &page);
            page_insert(mm->pgdir, page, addr, perm);
            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;
```

- 如果已经做好了交换的准备，才能做swap的处理，否则会输出 `no swap_init_ok but ptep is ..., failed\n` 然后直接返回。
- 页表项存在，但是仍然产生了page fault，说明需要把需要读取的内容从磁盘换入，因此使用 `swap_in(mm, addr, &page)`语句，把需要换入的那些内容从磁盘读到page中。
- 接下来，我们需要将这个新读入的page插入到mm->pgdir中，以建立物理页和虚拟内存的映射关系，因此使用 `page_insert(mm->pgdir, page, addr, perm)` 语句。
- 然后我们调用 `swap_map_swappable(mm, addr, page, 1)` 语句，把当前的新页设置为可以被替换。
- 最后使用 `page->pra_vaddr = addr` 把该物理页当前的虚拟地址绑定上。

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。**

  页目录表项和页表项在数据类型上都是64位无符号整数，他们只有54位有效，结构为：

  ```
  // Sv39 page table entry:
  // +----26---+----9---+----9---+---2----+-------8-------+
  // |  PPN[2] | PPN[1] | PPN[0] |Reserved|D|A|G|U|X|W|R|V|
  // +---------+----+---+--------+--------+---------------+
  ```

  它们的组成可以分为2个部分，前一部分代表物理页号，后一部分代表一些标记。 而二者的区别在于：页目录项的前一部分代表的是它所对应的大大页（一级页表）或大页（二级页表）的物理地址页号；而页表项的前一部分代表的是它所对应的物理页的物理地址页号。 关于他们有什么潜在的用处，可以去寻找使用了二者的函数，比如：

  ```
  static inline struct Page *pte2page(pte_t pte) {
     if (!(pte & PTE_V)) {
        panic("pte2page called with invalid pte");
     }
     return pa2page(PTE_ADDR(pte));
  }
  static inline struct Page *pde2page(pde_t pde) { //PDE_ADDR这个宏和
  PTE_ADDR是一样的
     return pa2page(PDE_ADDR(pde));
  }
  ```

  首先，我们可以看到pte和pde中那些标记发挥的作用。在 `if (!(pte & PTE_V))` 处，pte和PTE_V做了与操作，PTE_V是0x01，因此这一步就是在检验pte的最低位是否为1，结合上面对pte内容结构

的展示，也就是在**检验这个pte是否有效。**具体其他位的含义如下：

```
// page table entry (PTE) fields
#define PTE_V     0x001 // Valid
#define PTE_R     0x002 // Read
#define PTE_W     0x004 // Write
#define PTE_X     0x008 // Execute
#define PTE_U     0x010 // User
#define PTE_G     0x020 // Global
#define PTE_A     0x040 // Accessed
#define PTE_D     0x080 // Dirty
#define PTE_SOFT  0x300 // Reserved for Software
//一些其他用于检测标记位组合的函数
#define PAGE_TABLE_DIR (PTE_V)
#define READ_ONLY (PTE_R | PTE_V)
#define READ_WRITE (PTE_R | PTE_W | PTE_V)
#define EXEC_ONLY (PTE_X | PTE_V)
#define READ_EXEC (PTE_R | PTE_X | PTE_V)
#define READ_WRITE_EXEC (PTE_R | PTE_W | PTE_X | PTE_V)
#define PTE_USER (PTE_R | PTE_W | PTE_X | PTE_U | PTE_V)
```

然后，在函数的返回部分我们把pte作为PTE_ADDR函数的参数使用，我们具体关注到这个函数PTE_ADDR中。

```
// address in page table or page directory entry
#define PTE_ADDR(pte)   (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT -
PTE_PPN_SHIFT))
#define PDE_ADDR(pde)   PTE_ADDR(pde)
```

发现它就是通过与操作去除pte的低10位，然后左移了2位，以至于让它的结构和一个物理内存地址保持一致，这个函数的作用也就显现出来，即通过**页表项得到物理内存地址。** 这就是pte和pde的两个作用。

- **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?**

  1. 异常触发：当缺页服务例程尝试访问一个尚未在物理内存中加载的页面时，会触发页访问异常。
  2. 保存现场：硬件会保存当前执行线程的一些寄存器状态，如程序计数器（PC），栈指针（SP），以及一些其他相关的寄存器状态，以便之后能够恢复现场。
  3. 选择合适的异常处理程序：硬件会根据异常的类型，例如是页面不在内存中，还是只读页面被写入等，选择合适的异常处理程序，通常是操作系统内核中的缺页服务例程。
  4. 执行缺页服务例程：硬件会跳转到操作系统内核中的缺页服务例程，开始执行缺页处理代码。
  5. 恢复现场：一旦缺页处理完成，硬件会从缺页服务例程返回，恢复之前保存的现场信息。

- **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？** 有一定的对应关系，正如前面提到的：一级页表、二级页表、三级

页表都是用一个页来存储的。因此，三级页表的每个页表项都包含了某个page的物理地址，一级页表、二级页表的每个页目录项也都包含了某个page的物理地址，这个page就是用于存储该页表的。

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）。请在实验报告中简要说明你的设计实现过程。

设计实现过程简述：

1. 首先，我们仿照swap_fifo.c和swap_fifo.h创建swap_clock.c和swap_clock.h。
2. 在fifo的基础上，我们主要需要修改的就是_init_mm、_map_swappable和_swap_out_victim这三个函数。

```
    static int
    _clock_init_mm(struct mm_struct *mm)
    {
        // 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
        list_init(&pra_list_head);
        curr_ptr = &pra_list_head; //add：初始化curr_ptr指针（已定义为全局变
量）
        mm->sm_priv = &pra_list_head;
        cprintf("curr_ptr 0xffffffff%x\n",curr_ptr);
        return 0;
    }
```

```
    static int
    _clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
    {
        list_entry_t *entry=&(page->pra_page_link);
        assert(entry != NULL && curr_ptr != NULL);
        list_entry_t *head = (list_entry_t*) mm->sm_priv;
        // 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->prev, entry);//add: 将页面的visited标志置为1，表示该页
面已被访问
        page->visited = 1;
        cprintf("curr_ptr 0xffffffff%x\n",curr_ptr);
        //cprintf("hello");
        return 0;
    }
```

```
    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);
        while (1) {
            while(curr_ptr==head){
                curr_ptr = curr_ptr->next;
                // 遇到队头需要跳过
            }
            cprintf("curr_ptr 0xffffffff%x\n",curr_ptr);
            struct Page *tmp;
            // 用于存储当前的物理页
            list_entry_t *miduse;
            // 用于存储当前遍历到的list的指针
            tmp = le2page(curr_ptr,pra_page_link);
            // 将当前的列表指针
            if(!tmp->visited)
            {
                miduse = curr_ptr;
                curr_ptr = curr_ptr->next;
                *ptr_page = tmp;
                list_del(miduse);
                return 0;
            }
            else{
                tmp->visited=0;
                curr_ptr = curr_ptr->next;
                // 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
            }
            //cprintf("---hit q222head---\n");
        }
        return 0;
    }
```

- 比较 Clock 页替换算法和 FIFO 算法的不同。 Clock算法相比FIFO算法，多为Page页**设置了visited属性**，代表该页面是否被访问过，并且Clock算法使用的是**环形链表。**在选择具体替换出哪个页的时候，**二者的策略也不同。**FIFO算法不需要遍历链表，只需直接选定队头的那个页，而Clock算法需要进行链表的遍历，从某个位置开始进行扫描，先去尝试寻找visited标记为0的页，并且在扫描的过程中将visited标记为1的修改为标记为0。

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？

1. 好处：一个大页的页表映射方式，在查表阶段仅需一次内存访问，而分级页表需要更多的内存访问次数，比如说三级页表机制，就需要查一级页表、二级页表、三级页表来找到一个页表项。另外，一个大页的结构也更加简单，在页表的建立阶段，只需要进行一次页表的建立（内存分配）。
2. 坏处：一个大页的机制所占用的物理内存空间更大。比如以当前实验的机制来进行计算，一个大页需要的空间是$2^{9+9+9}*8=2^{30}$个字节，我们需要将这个1GB的大表整个加载到内存中，而三级页表结构下，我们查询一个项需要放入内存的页表大小是$(2^9+2^9+2^9)*8=3*2^{12}$个字节。因此，页表放在内存中就已经足够大了，更容易造成缺页异常，从而引发更多的访存操作。

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）

需写出有详细的设计、分析和测试的实验报告。

- 整体思路：当我们访问一个虚拟内存的时候，将它所在的页从原有链表删除，并放置在队列的尾端。
- 代码设计
    1. 在swap_lru.c文件中，创建以下函数，用于将最新被访问的页从原有链表删除，并放置在队列的尾端。

```
    static int
    _lru_time_update(struct mm_struct *mm, struct Page *page)
    {
       list_entry_t *head=(list_entry_t*) mm->sm_priv;
       list_entry_t *entry=&(page->pra_page_link);
       assert(entry != NULL && head != NULL);
       list_del(entry);
       list_add(head, entry);
       return 0;
    }
```

    2. 在swap.c文件中进行修改，在每次进行虚拟内存访问的时候，都调用check_content_part函数，其内容如下：

```
    static inline void
    check_content_part(pde_t* pgdir,struct mm_struct *mm,int la,int
what)
    {
       *(unsigned char *)la = what;
       *(unsigned char *)(la+0x10) = what;
       pte_t *ptepnew;
       struct Page *tmp;
       ptepnew = get_pte(pgdir,la, 0);
       tmp = pte2page(*ptepnew);
       sm->time_update(mm, tmp);
    }
```

    首先，它进行了两次虚拟内存访问操作，都是给某个虚拟地址赋值为what参数。然后，使用get_pte函数来获虚拟地址la所在的页表项，然后使用pte2page函数得到该页表项对应的物理页，然后再将该物理页进行1中提到的链表位置移动操作。
- 调试信息
    1. 在swap.c中修改check_swap函数如下：

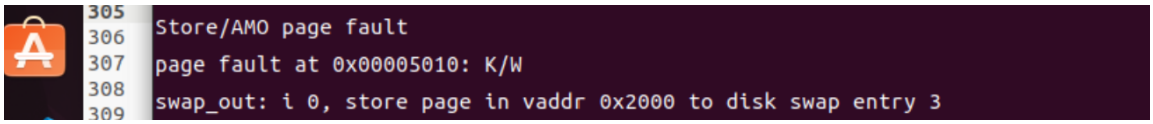
```
check_content_part(pgdir,mm,0x1000,0x0a);
check_content_part(pgdir,mm,0x2000,0x0b);
check_content_part(pgdir,mm,0x3000,0x0c);
check_content_part(pgdir,mm,0x4000,0x0d);
//tail-head: 4000-3000-2000-1000
```

```
check_content_part(pgdir,mm,0x1010,0x0a);
//tail-head: 1000-4000-3000-2000
check_content_part(pgdir,mm,0x5010,0x0e);
//tail-head: 5000-1000-4000-3000
check_content_part(pgdir,mm,0x3010,0x0c);
//tail-head: 3000-5000-1000-4000
check_content_part(pgdir,mm,0x2010,0x0b);
//tail-head: 2000-3000-5000-1000

cprintf("set up init env for check_swap over!\n");
// now access the virt pages to test  page relpacement algorithm

//restore kernel mem env
for (i=0;i<CHECK_VALID_PHY_PAGE_NUM;i++) {
      free_pages(check_rp[i],1);
}
.........
```

2. 前四个check_content_part的调用，都是在给0x1000、0x2000、0x3000、0x4000四个虚拟地址分配物理内存空间。所以应该会发生4次page fault，输出效果如下：

此时链表中应当是（tail-head）：4000-3000-2000-1000

3. 然后再访问0x1010处（它和0x1000是一页的），此时链表中应当是（tail-head）:1000-4000-3000-2000
4. 然后再访问0x5010处，我们没有给0x5000分配物理内存，所以会引发page fault，根据LRU算法，队头0x2000会被替换，输出效果如下：



此时链表中应当是（tail-head）：5000-1000-4000-3000

5. 然后再访问0x3010处，此时链表中应当是（tail-head）：3000-5000-1000-4000
6. 然后再访问0x2010处，由于在4中被替换出去，因此会引发page fault，根据LRU算法，队头0x4000会被替换，输出效果如下：