Lab3

实验目的：

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

实验内容：

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。

练习：

练习0——填写已有实验

手动将上次做的实验2的代码填入本次实验中代码中有“LAB2”的注释相应部分。

练习1：理解基于FIFO的页面替换算法（思考题）

1.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

1.1换入swap_in():

首先调用alloc_page()函数获取一个物理页面，然后调用get_pte()函数找到这个物理页面对应的页表项，再通过调用swapfs_read()函数将页表项的数据从硬盘读到内存上。

1.2换出swap_out()：

通过调用swap_out_victim()函数找到可以换出去的页面，然后调用swapfs_write()函数把要换出的物理页面写到硬盘上的交换区，写入失败，调用map_swappable()函数，将页面标记为可以被交换出去；写入成功，调用free_page()函数以及tlb_invalidate()函数，刷新TLB。

1.3中间过程page_insert()：

首先调用get_pte()函数找到对应页表项的位置，如果原先不存在，get_pte()会分配页表项的内存。然后调用page_ref_inc()函数将指向这个物理页面的虚拟地址增加了一个。再调用pte_create()函数构造页表项。最后调用tlb_invalidate()函数刷新TLB。

练习2——深入理解不同分页模式的工作原理（思考题）

1.解释get_pte()函数中为什么有两段代码如此相像

在通过虚拟地址获取对应的页表项上，sv32，sv39，sv48之间的差异是虚拟地址的位宽和各层级的索引数不同，但他们之间也有着共同点，那就是他们不同层级的操作在逻辑上高度相似。无论是SV32、SV39 还是 SV48，虚拟地址到物理地址的转换都是通过查找不同层级的页目录项和页表项来完成的。每一层的操作结构都是相似的，即检查当前目录或表项是否有效，若无效，则按需创建新的页表页并更新相应的项。因此，不同层级的操作在逻辑上高度相似，代码结构自然也会非常相像。

2.关于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的看法

我们小组经过讨论认为这样的写法可以满足基本的需求，但是从代码的可移植性、可维护性以及美观上考虑，不如拆分成两个函数。

那么有没有必要把两个功能拆开呢？我们小组认为没有必要。首先是刚才已经提到，这样的写法可以满足我们基本的要求，该有的功能都有了。其次是ucore本身就以代码少，而功能全著称，我们没有必要为了一些美观而对其进行拆分，导致代码量增多，甚至出现bug。

练习3——给未被映射的地址映射上物理页（需要编程）

设计实现过程：

1.明确需求：

通过观察未完成的do_pgfault（mm/vmm.c）函数、上文说明以及提供的英文注释。我们发现该函数已经实现了“给未被映射的地址映射上页表项”的功能，但是这个页表项还没有对应到具体的哪个物理页，也没有设置访问权限。这就是我们接下来要完成的任务。

2.代码编写：

首先是为这个页表项分配一个物理页。通过查询实验文档，我们了解到 `swap_in(mm, addr, &page)` 这个函数，该函数实现如下功能：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。这就实现了我们的第一个任务。

然后是将虚拟地址和刚得到的物理地址绑定起来。我们发现 `page_insert (mm->pgdir, page, addr, perm)` 可以通过修改页表项建立一个 `Page` 的 `phy addr` 与线性 `addr la` 的映射。

最后是为这个页设置页面可交换权限。利用 `swap_map_swappable(mm, addr, page, 0)` 函数完成我们的任务。

3.编写结果：

```
    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.
            // 检查 swap_in 是否成功
            if(swap_in(mm, addr, &page) !=0)
            {
                    cprintf("swap_in failed! \n");
                    goto failed;
            }

            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            // 检查映射是否成功
            if(page_insert(mm->pgdir, page, addr, perm) != 0)
            {
                    cprintf("page_insert failed! \n");
                    goto failed;
            }

            //(3) make the page swappable.
            swap_map_swappable(mm, addr, page, 0);
            page->pra_vaddr = addr;
            }
```

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。

1.明确PDE和PTE的组成部分并分析其作用

PDE：页表的物理地址、有效位、权限位。

- **页表的物理地址**：指向该页表的物理内存地址。
- **有效位（Valid bit）**：标记页目录项是否有效，若无效，表示该目录项指向的页表不可用。
- **权限位**：指定该页表是否可以进行读写操作、是否可以被用户访问等权限。

PTE：物理页框号、有效位、脏位、访问位、权限位。

- **物理页框号**：映射到的物理页的起始地址。

- **有效位（Valid bit）**：标记该页表项是否有效，若无效，表示虚拟页尚未映射到物理内存。
- **脏位（Dirty bit）**：表示该页面是否被修改过。即页面自从加载以来是否发生了写操作。
- **访问位（Accessed bit）**：表示该页面是否曾被访问过（读取或写入）。
- **权限位**：标识该页面的访问权限（如可读、可写等）。

2.描述其在页替换算法中的潜在用处

页替换算法，首先是要判断是否要发生页替换，而**有效位**的作用便是标记虚拟页是否映射到物理内存，**决定是否发生页替换行为**。其次是要在所有的页中找一个后面可能不再使用的页进行替换，根据“时间局部性原理”，我们认为最近没有被访问的页可能不会再被使用，故**访问位**可用来**决定是否将该页视为非“活跃”页**。再其次是要考虑换页时的速率问题，换页太慢自然优先级更低，换页慢的原因一般是数据发生了修改，即发生了写操作，故**脏位**可以**减少磁盘 I/O 操作的开销**。最后的**权限位**，不直接参与页替换的决策，但操作系统**可能会优先保留一些在权限上有特殊要求的页面**（如内核页），而选择更低权限的用户页进行替换。

上述描述均是基于PTE进行的。考虑到寻找页表的过程中一定会用到PDE，因此，上述描述也适用于PDE。

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？

我们小组经过讨论，认为针对这个新的页访问异常，硬件只需要按照正常处理异常的过程进行即可。在处理异常的过程中，硬件的任务是触发中断、保存状态、传递异常信息，以及在异常处理程序完成后恢复程序执行。

因此，硬件在处理缺页服务例程中的页访问异常时，主要会执行以下步骤：

- **产生中断**：硬件根据当前进程的页目录（Page Directory）和页表（Page Table）来检查虚拟地址的有效性。如果页表项指向的物理页面无效，CPU 就会触发缺页异常。
- **保存上下文**：硬件会自动保存当前的程序计数器（PC）、标志寄存器和其他必要的寄存器，以便稍后能够恢复到当前的执行点。
- **检查异常原因**：硬件会将异常的相关信息（如发生异常的虚拟地址）保存在 CPU 的某些寄存器中
- **将控制权交给操作系统**，让缺页服务例程处理该异常。
- **恢复执行**，在操作系统处理完缺页异常后，硬件会恢复程序的正常执行。

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？

Page 数组的每一项与页表中的页目录项和页表项之间的关系是间接的：页目录项指向页表，页表项指向物理页帧，而 Page 数组则表示这些物理页帧。

练习4——补充完成Clock页替换算法（需要编程）

设计实现过程：

1.明确需求：

通过观察未完成的Clock页替换算法、上文说明以及提供的英文注释。我们发现该算法仍需完成 `_clock_init_mm` 、 `_clock_map_swappable` 以及 `_clock_swap_out_victim` 三个函数。其中，每个函数需要干什么，都给出了明确的注释。

2.代码编写：

首先是 `_clock_init_mm` 函数，我们需要对已有的链表进行初始化工作。前面我们学过 `list_init()` 函数可以用来初始化链表；通过赋值操作将指针 `curr_ptr` 指向 `pra_list_head` ；将mm的私有成员指针指向 `pra_list_head` ，mm的私有成员指针有多个，但和交换页相关的便是 `sm_priv` （the private data for swap manager），因此只要将 `pra_list_head` 的地址赋值给 `sm_priv` 即可。

然后是 `_clock_map_swappable` 函数，将页面page插入到页面链表 `pra_list_head` 的末尾， `_clock_init_mm` 函数中已经提到，用“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”，故使用 `mm->sm_priv` 代替 `pra_list_head` 。另外page本身是无法插进去的，要用list_entry_t格式变量，正好第一行给出了这个变量，因此 `list_add(mm->sm_priv, entry)` 完成任务，同时将visited标志置为1。

最后是 `_clock_swap_out_victim` 函数，遍历页面链表 `pra_list_head` ，由于现在已经是while循环了，我们首先检验一下链表是否为空，同时确保 `curr_ptr` 指向有效节点。然后我们将当前变量不断前移即可 `curr_ptr = list_prev(curr_ptr);` 获取当前页面对应的Page结构指针，这使用前面学习的 `le2page ()` 函数即可； 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，只需要一个if语句即可；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， `list_del()` 函数解决删除问题，返回就更简单了，现在的page本身就是所需的格式，直接赋值结束循环即可。

3.编写结果：

```
1       // 初始化pra_list_head为空链表
2       list_init(&pra_list_head);
3       // 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
4       curr_ptr=&pra_list_head;
5       // 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
6       mm->sm_priv = &pra_list_head;
7   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ~
```

```
    /*LAB3 EXERCISE 4: 2210751*/
    // 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_entry_t *head=(list_entry_t*) mm->sm_priv;

    list_add(head, entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited = 1;
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
while (1) {
        /*LAB3 EXERCISE 4: 2210751*/
        // 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

        //空链表检查
        list_entry_t *entry = list_prev(head);
        if(entry == head)
        {
                *ptr_page = NULL;
                break;
        }
        //确保curr_ptr 始终指向有效的页面节点
        if(curr_ptr == head)
        {
                curr_ptr = curr_ptr ->prev;
        }
        //获取页面指针
        struct Page *page = le2page(curr_ptr, pra_page_link);

        if(page->visited == 0)
        {
                cprintf("curr_ptr %p\n",curr_ptr);
            list_del(curr_ptr);
            *ptr_page = page;
            break;
        }
        else
        {
                page->visited = 0;
        }

```

```
52          curr_ptr = list_prev(curr_ptr);
53
54      }
```

比较Clock页替换算法和FIFO算法的不同

FIFO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。

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。

最大的不同在于FIFO算法直接淘汰在内存中最久的页，而Clock算法淘汰在内存中最近没有被访问过的最久的页。

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）

1.”一个大页“ 的页表映射方式相比分级页表，有什么好处，有什么坏处？

好处：

1. **简单性与效率**：
   - **减少页表查找次数**：在分级页表中，虚拟地址需要通过多个级别的页表进行转换（例如，二级页表需要两次查找），而一个大页方式将虚拟地址映射到物理地址时，通常只需要一次查找。这可以显著减少内存访问延迟。
   - **减少页表大小**：如果使用一个单独的大页表，而不是多个层级的页表，操作系统和硬件的管理开销较小。例如，较大的页表会减少页表项的数量，减少了页表存储空间的需求。

2. **减少TLB缺失**：
    - 使用大页（如2MB或更大）可以增加每个TLB（Translation Lookaside Buffer）条目的映射数量，减少TLB缺失的次数，因为大页能够映射更多的内存区域。较大的页面能够更有效地利用TLB缓存，从而提高内存访问速度。
3. **减少内存碎片**：
    - 在大页映射的情况下，内存分配更集中，可能减少小页面的分散存储，降低内存碎片的产生。

坏处：

1. **内存浪费**：
    - 大页的最直接问题是 **内存碎片**。因为大页需要一整块连续的物理内存区域，所以在某些情况下，内存分配可能不够灵活，导致 **内存利用率低**。如果程序并不需要这么大的内存块，使用大页就会浪费大量空间。
    - 例如，如果一个应用程序只需要1KB的内存，但由于使用了一个2MB的大页，它将会占用2MB的物理内存，造成内存浪费。
2. **硬件和操作系统的支持问题**：
    - **硬件要求**：大页的支持要求硬件的TLB和页表项必须能处理更大的页大小。虽然现代CPU通常支持大页（如2MB、1GB等），但这仍然取决于硬件是否能有效支持大页。对于不支持大页的系统，使用大页可能会带来性能的下降。
    - **操作系统支持**：操作系统需要更复杂的机制来处理大页。大页的分配和管理可能导致操作系统在某些场景下无法灵活高效地调整内存的使用情况。
3. **较大的页表项**：
    - 大页映射需要较大的页表项（例如，2MB页面需要12位的页表项），这可能导致单个页表的内存消耗增加。虽然从单一页表的角度来看可能较为简单，但在高内存需求的系统中，这样的做法可能导致页表本身成为内存的消耗源。
4. **不灵活的内存管理**：
    - 如果系统使用较大的页表映射，动态分配内存变得不那么灵活。分配小内存块时，可能因为需要对齐或申请整个大页而产生浪费，而无法有效支持内存的细粒度控制。
5. **调度复杂性增加**：
    - 大页映射可能对操作系统的内存管理算法带来一定的挑战。尤其是当大页映射需要切换到物理内存时，会影响 **页面交换（swap）** 和 **内存回收** 等机制的效率。操作系统可能需要进行额外的管理，以避免频繁的页表更新或大页回收的开销。

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）

算法基本原理：

LRU算法基于一个简单的假设：

> “最近使用的页面将来更可能被访问，而很久未使用的页面很可能在未来也不会被访问。”

LRU算法会记录每个页面上次被访问的时间，当需要换页时，选择**最久未被访问的页面**进行替换。

使用的数据结构：

用一个双向链表维护页面访问的顺序，链表头部是最近访问的页面，尾部是最久未访问的页面。

实现过程：

根据lru的基本思想，同时参考swap_fifo.c和swap_clock.c相关代码，我们完成 `_lru_init_mm` 、 `_lru_map_swappable` 、 `_lru_swap_out_victim` 、 `_lru_check_swap` 、 `_lru_init` 、 `_lru_set_unswappable` 、 `_lru_tick_event` 函数以及 `swap_manager swap_manager_lru` 结构体。我们还需要实现一个访问页面后，更新页面的操作，我们将该操作封装为一个函数 `_lru_update_pages` 。

函数 `_lru_update_pages` 操作主要为，遍历找到对应的页面，然后将其放置在head的后一节点，表示最新访问的。

代码如下：

```
//更新页面，在访问时将其更新
static int _lru_update_pages(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);

    // 遍历链表，检查是否存在访问的页面
    curr_ptr = head->next;
    while(curr_ptr!=head)
    {
        struct Page *curr_page = le2page(curr_ptr, pra_page_link);
        if (curr_page == page)
        {
            break;  // 找到页面，退出循环
        }
        curr_ptr = curr_ptr->next;
    }

    // 页面不存在于链表中
    if (curr_ptr == head)
    {
        return 0;
    }
```

```
26
27      // 页面已经在链表头部
28      if (entry->prev == head)
29      {
30          return 0;
31      }
32
33      // 从链表中删除页面
34      list_del(entry);
35
36
37      // 将页面插入到链表头部
38      list_add(head, entry);
39      return 0;
40  }
```

但是，我们最初的思想是将该操作放置在 `_lru_map_swappable` 函数中，在 `list_add(head, entry);` 之前，但是使用 `make qemu` 命令就会发生报错，将该操作注释或者放在 `list_add(head, entry);` 之后，就会正常运行。经过上网查阅发现，该实验只是实现了简单的页面置换，并未涉及进程相关的操作，也就没有实现页面的访问，所以加入这个操作则会导致其发生报错。