操作系统实验报告

实验名称：缺页异常和页面置换

组号：46　　小组成员：王彬 张泽睿 岳建新

一、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。

二、实验过程

1.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。

换入换出流程

简述整个虚拟内存管理的过程：当发生缺页异常时，出现异常的地址会传入 `get_pte` 函数获取或创建对应虚拟地址映射过程中的页表和页表项：

1. `do_pgfault`：开始处理缺页异常，根据 `get_pte` 得到的页表项内容确定页面是需要创建还是需要换入。
    - **若得到的页表项为 0**，表示该页从未被分配，此时使用：
        - `pgdir_alloc_page`：分配一个物理页。
        - `page_insert`：建立虚拟地址与物理页的映射。
        - 注意：在 `alloc_pages` 中增加一个循环，当页面数量不足时，使用 `swap_out` 函数将对应数量的页面换出。
            - 在 `swap_out` 中，调用页面置换管理器的 `swap_out_victim` 按照 FIFO 策略选择要换出的页面，并使用 `swapfs_write` 将页面内容写入"硬盘"中。
    - **若得到的页表项不为 0**，表示该地址对应的页之前已被换出，此时需要：
        - 调用 `swap_in`：将页面内容从"硬盘"中通过 `swapfs_read` 读入对应的物理页。
        - 使用 `page_insert` 将虚拟地址与新分配的物理页建立映射。
        - 调用 `swap_map_swappable`：根据不同策略维护页面置换，确保页面置换能正确执行。

总体流程描述：

当执行到 `*addr = ...` 时，首先触发缺页异常，进入 `trap.c`，依次执行并调用核心处理函数 `do_pgfault`。`do_pgfault` 通过获取 `vma` 和 `mm` 等管理结构，调用 `get_pte` 获取或创建 `addr` 对应的物理页和页表项。如果页表项为 0，则分配新页并建立映射；如果页表项不为 0，则将原页换出到磁盘，并将所需页换入内存，最后建立新的映射关系，完成缺页处理。

函数的解释：

1. `do_pgfault`：整个缺页处理流程的开始，根据 `get_pte` 得到的页表项的内容确定页面是需要创建还是需要换入。
2. `pgfault_handler`：处理页面错误，调用 `do_pgfault` 来处理具体的页面错误。
3. `check_mm_struct`：检查内存管理结构，确保内存管理结构有效。
4. `find_vma`：查找包含指定地址的虚拟内存区域（VMA），返回包含指定地址的 VMA。
5. `get_pte`：获取或创建虚拟地址 `la` 对应的页表项（PTE），检查页目录项和页表项是否存在，如果不存在则分配新的页面并初始化。
6. `pgdir_alloc_page` -> `alloc_page`：分配一个新的物理页面，从物理内存中分配一个空闲页面，并返回该页面的指针。
7. `page_insert`：将页面插入到页表中，将页面的物理地址和标志位写入页表项，并增加页面的引用计数。
8. `free_page`：释放一个物理页面，将页面标记为空闲，并减少页面的引用计数。
9. `swap_map_swappable`：将页面标记为可换出，将页面添加到可换出的页面列表中。
10. `assert`：断言宏，用于调试，检查条件是否为真，如果为假则触发断言失败。
11. `memset`：将内存区域的前 `n` 个字节设置为指定的值 `c`，初始化内存区域。
12. `pte_create`：创建一个页表项，根据物理页号和标志位创建一个页表项。
13. `page2ppn`：将页面结构体指针转换为物理页号，计算并返回页面对应的物理页号。
14. `swap_out`：将页面换出到交换区，选择一个要换出的页面，将其内容写入交换区，并更新页表项。
15. `swap_in`：将页面从交换区换入内存，从交换区读取页面内容，分配新的物理页面，并更新页表项。
16. `ide_read_secs`：从磁盘读取扇区，将指定扇区的数据读取到内存中。
17. `ide_write_secs`：将扇区写入磁盘，将内存中的数据写入到指定的磁盘扇区。
18. `swap_out_victim`：选择一个要换出的页面，根据 FIFO 算法选择一个要换出的页面。
19. `tlb_invalidate`：使 TLB 中的对应条目失效，在页面换出或换入时，确保 TLB 中的旧条目失效，以便新的映射生效。

2. 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。

1. 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。

在`get_pte()`函数中，通常会有两段类似的代码用于遍历页表层级并获取对应的页表项（PTE）。这种相似性主要源于不同分页模式（如sv32、sv39、sv48）在页表结构上的共性，即它们都是通过多级页表来实现虚拟地址到物理地址的映射。

- **分页模式的相似性**：
  - **多级页表架构**：无论是sv32、sv39还是sv48，都采用多级页表结构（如二级、三级或四级），通过逐级索引来定位最终的页表项。
  - **地址分割方式**：各个分页模式都会将虚拟地址划分为不同的部分，用于索引不同层级的页表。例如，sv32使用两级页表，sv39使用三级页表，sv48则使用四级页表，但基本的索引方式类似。
- **sv32，sv39，sv48的不同**：
  - 地址位数和虚拟地址空间大小
    Sv32 使用 32 位虚拟地址，支持 4 GiB 的虚拟地址空间，适用于简单的系统，如嵌入式设备。Sv39 和 Sv48 则使用更宽的虚拟地址位数，分别为 39 位和 48 位，支持更大的虚拟地址空间，Sv39 支持 512 GiB，而 Sv48 支持 256 TiB，适用于更复杂的系统或服务器。
  - 页表层级
    Sv32 使用两级页表结构，即页目录和页表，以减少嵌套的层级来适配较小的虚拟地址空间需求。Sv39 和 Sv48 使用三级和四级页表结构，分别增加了中间页表层级。Sv39 有页目录（第一级）、页中间目录（第二级）和页表（第三级），而 Sv48 增加了一层页顶目录（第四级），这些额外的层级允许更灵活的内存分配和映射。
  - 物理页面大小
    Sv32、Sv39 和 Sv48 都支持 4 KiB 的基础页面大小，但 Sv39 和 Sv48 还可以支持更大页面（2 MiB 和 1 GiB），通过在不同层级的页表项中直接映射来提高管理效率。Sv32 仅支持基础的 4 KiB 和较大的 1 MiB 页面。
  - 地址转换的效率和复杂性
    随着页表层级的增加，Sv39 和 Sv48 的地址转换相较于 Sv32 更加复杂，需要更多步才能完成转换，但这种设计换取了对大规模地址空间的支持。Sv32 的两级页表查找速度较快，但其地址空间有限，不适合需要较大内存的应用。
- **代码相似性的原因**：
  - **统一的查找逻辑**：尽管分页模式的层级和地址位宽不同，但查找页表项的基本逻辑是一致的，即逐级索引页目录和页表，最终获取PTE。
  - **可复用的函数结构**：为了代码的可维护性和复用性，`get_pte()`函数在实现时会采用类似的代码结构，区别仅在于分页模式的具体参数（如页表层级数、每级的索引位数等）。

因此，这两段代码在不同分页模式下表现出高度相似性，是因为它们遵循相同的多级页表查找逻辑，只是在具体的分页模式参数上有所不同。

2. 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？

将页表项的查找和分配合并在`get_pte()`函数中有其优点和缺点，具体如下：

- **优点**：
  - **简化调用接口**：调用者只需调用一个函数即可完成查找或分配，简化了函数的使用。

    - **提高代码的紧密性**：在需要查找并可能分配页表项的场景下，合并处理可以减少函数调用的次数，提高执行效率。
    - **减少重复代码**：将查找和分配逻辑结合，可以避免在不同地方编写重复的代码，实现代码复用。
- **缺点**：
    - **功能耦合**：查找和分配是两个不同的操作，将它们合并会增加函数的复杂性，使其职责不够单一，违背单一职责原则。
    - **灵活性降低**：有时仅需要查找页表项而不需要分配新的页面，此时合并的函数可能会引入不必要的分配操作，导致资源浪费。
    - **测试和维护困难**：功能耦合使得单独测试查找或分配变得困难，增加了维护的复杂度。
- **建议**：
    - **拆分功能**：将页表项的查找和分配分成两个独立的函数，例如`find_pte()`和`allocate_pte()`。这样可以根据不同的需求选择调用对应的功能，提高代码的灵活性和可维护性。
    - **保持接口简洁**：为常见的查找或查找+分配操作提供高层接口，如`get_pte()`，内部调用`find_pte()`和`allocate_pte()`，实现功能复用的同时保持接口的简洁性。

综上所述，虽然将页表项的查找和分配合并在`get_pte()`函数中在某些情况下简化了调用，但从代码设计的角度来看，拆分这两个功能会提高代码的可维护性和灵活性。因此，建议将查找和分配功能拆分为独立的函数，并通过高层接口实现常见的操作需求。

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。

设计实现过程

在`do_pgfault`函数中，我们需要处理缺页异常，为未被映射的地址分配物理页并建立映射。具体步骤如下：

1. **查找 VMA**：根据缺页地址查找对应的虚拟内存区域（VMA），获取访问权限。
2. **获取或创建页表项**：使用`get_pte`函数获取或创建页表项。
3. **分配物理页**：如果页表项为空，使用`pgdir_alloc_page`分配新的物理页并建立映射。
4. **处理交换条目**：如果页表项为交换条目，使用`swap_in`将页面从交换区换入内存，并使用`page_insert`建立映射。
5. **设置页面可交换**：使用`swap_map_swappable`设置页面为可交换。

补充代码

```
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;
    }

    /* IF (write an existed addr ) OR
     *    (write an non_existed addr && addr is writable) OR
     *    (read  an non_existed addr && addr is readable)
     * THEN
     *    continue process
     */
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    /*
    * Maybe you want help comment, BELOW comments can help you finish the code
    *
    * Some Useful MACROs and DEFINEs, you can use them in below implementation.
    * MACROs or Functions:
    *   get_pte : get an pte and return the kernel virtual address of this pte
for la
    *             if the PT contians this pte didn't exist, alloc a page for PT
(notice the 3th parameter '1')
    *   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
    * DEFINES:
    *   VM_WRITE  : If vma->vm_flags & VM_WRITE == 1/0, then the vma is
writable/non writable
    *   PTE_W           0x002                   // page table/directory entry
flags bit : Writeable
    *   PTE_U           0x004                   // page table/directory entry
flags bit : User can access
    * VARIABLES:
    *   mm->pgdir : the PDT of these vma
    *
    */


    ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                         //PT(Page Table) isn't existed, then
                                         //create a PT.
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {
        /*LAB3 EXERCISE 3: YOUR CODE
        * 请你根据以下信息提示，补充函数
```

```
            * 现在我们认为pte是一个交换条目，那我们应该从磁盘加载数据并放到带有phy addr的页面，
            * 并将phy addr与逻辑addr映射，触发交换管理器记录该页面的访问情况
            *
            *  一些有用的宏和定义，可能会对你接下来代码的编写产生帮助(显然是有帮助的)
            *  宏或函数:
            *    swap_in(mm, addr, &page) : 分配一个内存页，然后根据
            *    PTE中的swap条目的addr，找到磁盘页的地址，将磁盘页的内容读入这个内存页
            *    page_insert :  建立一个Page的phy addr与线性addr la的映射
            *    swap_map_swappable :  设置页面可交换
            */
            if (swap_init_ok) {
                struct Page *page = NULL;
                // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
                //(1) According to the mm AND addr, try
                //to load the content of right disk page
                //into the memory which page managed.
                //(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);//换入缺失的页
                page_insert(mm->pgdir,page,addr,perm);//页插入到管理的页表中
                swap_map_swappable(mm,addr,page,1);//设置页面可交换
                page->pra_vaddr = addr;
            } else {
                cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
                goto failed;
            }
        }

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

代码首先通过`if (*ptep == 0)`判断获取到的页表项ptep是否为空项，若为空项，则分配对应的物理页给这个页表项。

如果不是空项，说明物理页不存在于内存中，而在磁盘中，需要进行页交换处理。那么使用swap_in函数来将需要的物理页读入内存(如果内存不足，自动调用swap_out函数换出，最终一定能成功将该物理页换入)，然后使用page_insert来建立页表项到页之间的映射，最后把其可交换属性设置为真，插入FIFO的队列中。

问题回答

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。

页目录项（PDE）和页表项（PTE）包含物理页号（PPN）和控制位（如有效位、读/写/执行权限位、用户/内核模式位等）。这些信息在页替换算法中具有重要作用：

- **PDE 和 PTE 中的物理页号（PPN）**：在实现换入和换出操作时，操作系统需要知道某个页面的物理地址，以便更新页表并执行物理内存的读 写操作。页目录项和页表项中保存的物理页地址为此提供了必要的信息。

- **页表项的访问标志位**：在 LRU 算法中，访问位（PTE_A）可以帮助操作系统判断哪些页面最近没有被访问，从而选择它们作为被换出的候选。脏位（PTE_D）则帮助操作系统决定是否需要将修改过的页面写回磁盘。
- **访问权限位（PTE_R、PTE_W、PTE_X）**：用于设置页面的访问权限，确保页面访问的安全性。
- **用户/内核模式位（PTE_U）**：区分用户态和内核态的页面访问权限。
- **页目录项和页表项的有效位（PTE_V）**：在页面置换过程中，操作系统可以通过检查页表项的有效位来确定页面是否已加载到内存。如果页表项 无效，操作系统可能会将页面标记为交换候选，或者将其从交换空间加载回内存。

以上信息可以帮助操作系统按需求实现不同的页面替换算法,如访问标志位可以帮助实现 LRU 算法,以下是对 LRU 算法的简介：

LRU（Least Recently Used）算法是一种常用的缓存替换策略，主要用于内存管理、数据库和操作系统等需要缓存的场景。其核心思想是，如果一个数据最近被访问过，那么在短期内它再次被访问的概率较大；而那些长时间未被访问的数据，再次被访问的概率相对较低。因此，在缓存满的情况下，LRU 算法选择淘汰最久未被访问的数据，以便腾出空间给新数据。

具体的 LRU 算法我们已经在 challenge 中完成了实现，详细内容可见报告的最后部分。

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?

出现页访问异常时：

1. 首先使用 `SAVE_ALL` 把 `trapFrame` 结构体放到栈顶， `trapFrame` 中保存了通用寄存器和中断处理相关的 `CSR` 。
2. 异常的指令的地址存入 `sepc` 寄存器，异常或陷阱相关的具体信息存如 `stval` 寄存器。
3. 根据 `stvec` 跳入中断处理程序中进行中断处理。

   以上内容为 Lab1 的知识部分，接下来会：
4. 跳入 `trap.c` 中调用 `exception_handler()` ，进而根据 `scause` 的值调用 `pgfault_handler` 进行异常处理。

简要的总结，当缺页服务例程在执行过程中访问内存，出现页访问异常时，硬件需要执行以下操作：

- **保存处理器状态**：将当前处理器状态（如程序计数器、寄存器值等）保存到陷阱帧（trapframe）中。
- **设置异常原因**：在异常原因寄存器（如 `scause` ）中记录异常类型和原因。
- **跳转到异常处理入口**：根据异常向量表跳转到操作系统定义的异常处理入口，通常是缺页异常处理函数。

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?

数据结构 `Page` 的全局变量（数组）中的每一项与页表中的页目录项（PDE）和页表项（PTE）有对应关系：

- **Page 结构体**：表示物理内存中的一个页面，包含页面的元数据（如引用计数、标志位等）。
- **页表项（PTE）**：包含物理页号（PPN），指向 `Page` 结构体对应的物理页面。

- **对应关系**：页表项中的物理页号（PPN）与 `Page` 结构体数组的索引对应，通过页表项可以找到对应的 `Page` 结构体，从而访问和管理物理页面。

以下是对该小问的一些分析：

page` 结构体的定义如下:

```
struct Page {
    int ref;                        // page frame's reference counter
    uint_t flags;                 // array of flags that describe the status of
the page frame
    uint_t visited;
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
    list_entry_t pra_page_link;     // used for pra (page replace algorithm)
    uintptr_t pra_vaddr;            // used for pra (page replace algorithm)
};
```

可以通过以下运算得到物理页号和起始的物理地址：

```
// page2ppn
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }

// page2pa
static inline uintptr_t page2pa(struct Page *page)
    {return page2ppn(page) << PGSHIFT; }
```

- 每一个页表所占用的空间刚好为一个页的大小
- 通过 PTE ，操作系统能够找到物理页面，从而对该物理页面进行管理。而 Page 结构体则提供了关于该物理页面的元数据和状态。
- 通过物理地址可以确定物理页号，从而找到对应的Page结构体。

4.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 页替换算法页面（mm/swap_clock.c）。请在实验报告中简要说明你的设计实现过程。

实现过程：

完成如下各个函数;

_clock_init_mm

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4 2212784*/
     // 初始化pra_list_head为空链表
     list_init(&pra_list_head);
     // 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
     curr_ptr = &pra_list_head;
     // 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
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

根据提示，使用list_init初始化pra_list_head为空链表，然后令curr_ptr指向表头，将mm的私有成员指针指向pra_list_head即可。

_clock_map_swappable

```
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);
    //record the page access situlation
    /*LAB3 EXERCISE 4: 2212784*/
    // 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_entry_t *head=(list_entry_t*)mm->sm_priv;
    list_add(head->prev, entry);
    page->visited=1;
    return 0;
}
```

每次将新加入的可交换的表项插入到头节点前方，遍历从头节点后侧开始

_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: 2212784*/
        // 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
        if (curr_ptr == &pra_list_head) {
            curr_ptr = list_next(curr_ptr);
        }
        // 获取当前页面对应的Page结构指针
        struct Page *ptr = le2page(curr_ptr, pra_page_link);
        if(ptr->visited == 1) {
            ptr->visited = 0;
            curr_ptr = list_next(curr_ptr);
        }
        else{
            cprintf("curr_ptr %p\n",curr_ptr);
            curr_ptr = list_next(curr_ptr);
            list_del(curr_ptr->prev);
            *ptr_page = ptr;
            break;
        }
    }
    return 0;
}
```

• 比较 Clock 页替换算法和 FIFO 算法的不同。

对于页面链表 `pra_list_head` 来说，Clock 页替换算法在每次加入新页时添加到链表的末尾，而执行换出页面时从全局变量当前指针 `curr_ptr` 开始查找最早未被访问的页面。`_clock_swap_out_victim` 函数中的循环意图即为从当前指针开始查找，如果该页面已被访问则将 `visited` 状态设置为 0 起到重置作用，如果该页面未被访问，则调用 `list_del` 函数将该页面从页面链表中删除，并将该页面指针赋值给参数 `ptr_page`（`*ptr_page = ptr`）作为换出页面。这样即使链表中所有页之前都被访问过，也会按照该逻辑遍历一圈后删除 `curr_ptr` 指向的页面。

具体来说，Clock 页替换算法的步骤如下：

1. **遍历页面链表**：从当前指针 `curr_ptr` 开始遍历页面链表 `pra_list_head`。
2. **检查页面访问状态**：
    - 如果当前页面已被访问（`visited` 标志为 1），则将 `visited` 标志置为 0，表示该页面已被重新访问。
    - 如果当前页面未被访问（`visited` 标志为 0），则将该页面从页面链表中删除，并将该页面指针赋值给参数 `ptr_page` 作为换出页面。
3. **更新当前指针**：将 `curr_ptr` 更新为下一个页面，继续遍历。

与之相反的是 FIFO 算法，其每次添加新页时添加到链表的头部，每次换出页面时调用 `list_prev` 找链表头的前一个节点（因为是双向链表，直接找到链表尾）来进行换出。如果该页面不是链表头（是链表头意味着只有一个元素，按照换出逻辑来说不存在该情况），则将该页面赋值给参数 `ptr_page` 作为换出页面。

具体来说，FIFO 页替换算法的步骤如下：

1. **添加新页**：每次添加新页时，将新页添加到链表的头部。
2. **换出页面**：每次换出页面时，调用 `list_prev` 找到链表头的前一个节点（即链表尾），将该页面从链表中删除，并将该页面指针赋值给参数 `ptr_page` 作为换出页面。

注意到 `curr_ptr` 的本质作用也是指向最老的页面，不过其会随着链表的插入删除变化，而不是每次都从链表头遍历。当然更关键的是 Clock 考虑了页面的访问情况，而不是像 FIFO 一样粗暴地驱逐最早进入的页面。

FIFO算法还有一个致命的缺点，就是内存越大反而发生缺页异常的次数越多。

5. 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?

采用"一个大页"的页表映射方式，即使用大页（如 2MB 或 1GB 页）而不是传统的分级页表（如 4KB 页），在某些情况下具有一定的优势和劣势。以下是详细的比较：

好处和优势

1. **减少页表项数量**：
   - **优势**：大页映射方式可以显著减少页表项的数量，因为每个大页覆盖的地址范围更大。例如，一个 2MB 页可以覆盖 512 个 4KB 页的地址范围。
   - **好处**：减少页表项数量可以降低页表的存储开销，减少内存占用。
2. **减少页表查找开销**：
   - **优势**：大页映射方式可以减少页表查找的层级，从而降低页表查找的开销。例如，使用 1GB 页时，可以减少多级页表查找的次数。
   - **好处**：减少页表查找开销可以提高内存访问的效率，提升系统性能。
3. **提高 TLB 命中率**：
   - **优势**：大页映射方式可以提高 TLB（Translation Lookaside Buffer）的命中率，因为每个 TLB 条目可以覆盖更大的地址范围。
   - **好处**：提高 TLB 命中率可以减少 TLB 缺失的次数，进一步提升内存访问的效率。

坏处和风险

1. **内存浪费**：
   - **风险**：大页映射方式可能导致内存浪费，因为每个大页的大小固定，如果实际使用的内存不足一个大页的大小，未使用的部分将被浪费。
   - **坏处**：内存浪费可能导致系统内存利用率降低，特别是在内存资源紧张的情况下。
2. **内存碎片化**：
   - **风险**：大页映射方式可能导致内存碎片化，因为大页需要连续的物理内存，如果系统内存碎片较多，可能难以找到足够大的连续内存块来分配大页。
   - **坏处**：内存碎片化可能导致内存分配失败，影响系统稳定性和性能。
3. **灵活性降低**：
   - **风险**：大页映射方式的灵活性较低，因为大页的大小固定，无法根据实际需求灵活调整。
   - **坏处**：灵活性降低可能导致内存管理的复杂性增加，难以适应多样化的应用场景。
4. **页表更新复杂性增加**：
   - **风险**：大页映射方式在页表更新时可能需要更多的操作，因为大页覆盖的地址范围更大，更新一个大页可能影响更多的内存区域。
   - **坏处**：页表更新的复杂性增加可能导致系统开销增加，影响性能。
5. **换入换出用时增加，降低运行速率**：

- 使用一个大页进行映射意味着在发生缺页异常和需要页面置换时需要把整个大页的内容（在Sv39下即为1GiB）全部交换到硬盘上，在换回时也需要将所有的内容一起写回。在物理内存不足或者进行运行数量多是会频繁换入换出，造成运行速率较低。

6. **安全隐患**：
   - 大页会暴漏更多内存给恶意代码，会增加安全风险。

总结

采用"一个大页"的页表映射方式在减少页表项数量、减少页表查找开销和提高 TLB 命中率方面具有明显的优势，但也存在内存浪费、内存碎片化、灵活性降低和页表更新复杂性增加，运行速率降低，产生安全隐患的风险。在实际应用中，需要根据具体的应用场景和系统需求权衡利弊，选择合适的页表映射方式。

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）

首先在 `swap.c` 中增加

```
#include <swap_lru.h>
```

并将

```
sm = &swap_manager_clock;
```

改为

```
sm = &swap_manager_lru;
```

接下来完成LRU策略：

`lru.h`:

```
#ifndef __KERN_MM_SWAP_LRU_H__
#define __KERN_MM_SWAP_LRU_H__

#include <swap.h>
extern struct swap_manager swap_manager_lru;

#endif
```

`lru.c`:

```
#include <defs.h>
#include <riscv.h>
#include <stdio.h>
#include <string.h>
#include <swap.h>
#include <swap_lru.h>
#include <list.h>
```

```c
extern list_entry_t pra_list_head;

static int
_lru_init_mm(struct mm_struct *mm)
{
    list_init(&pra_list_head);
    mm->sm_priv = &pra_list_head;
    return 0;
}

static int
_lru_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);

    // 将最近访问的页面添加到链表头部
    list_add(head, entry);
    return 0;
}

static int
_lru_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);

    // 选择最久未使用的页面进行替换
    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;
}

static void
chect_list(uintptr_t addr) {  // 检查当前页是否在链表中，如果在就放在链表头
    list_entry_t *head = &pra_list_head, *le = head;

    while ((le = list_prev(le)) != head) {
        struct Page *curr = le2page(le, pra_page_link);
        if (curr->pra_vaddr == addr) {
            list_del(le);          // 删除找到的页
            list_add(head, le);    // 将页移到链表头部
            return;
        }
    }
}
```

```
static void
printlist() {   // 打印链表
    cprintf("--------list-head----------\n");
    list_entry_t *head = &pra_list_head, *le = head;
    while ((le = list_next(le)) != head)
    {
        struct Page* page = le2page(le, pra_page_link);
        cprintf("vaddr: %x\n", page->pra_vaddr);
    }
    cprintf("--------list-end----------\n");
}

static void
write_and_check(uintptr_t addr, unsigned char value) { // 写入一个字节的数据，对虚拟地址的写操作
    cprintf("write Virt Page 0x%x in lru_check_swap\n", addr);
    chect_list(addr);
    *(unsigned char *)addr = value;
}

static int
_lru_check_swap(void) {
    uintptr_t test_addrs[] = {0x3000, 0x1000, 0x4000, 0x2000, 0x5000};//vma里面没有0x6000
    unsigned char values[] = {0x0c, 0x0a, 0x0d, 0x0b, 0x0e};

    for (int i = 0; i < 5; i++) {
        write_and_check(test_addrs[i], values[i]);
        printlist();
    }
    write_and_check(0x1000, 0x0a);
    printlist();

    return 0;
}

static int
_lru_init(void)
{
    return 0;
}

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

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

struct swap_manager swap_manager_lru =
{
    .name            = "lru swap manager",
    .init            = &_lru_init,
```

```
    .init_mm          = &_lru_init_mm,
    .tick_event       = &_lru_tick_event,
    .map_swappable    = &_lru_map_swappable,
    .set_unswappable = &_lru_set_unswappable,
    .swap_out_victim = &_lru_swap_out_victim,
    .check_swap       = &_lru_check_swap,
};
```

接下来是对`lru`策略的一些解释：

`lru`策略维护了一个访问链表，链表头是最近访问的页，链表尾是最远访问（没有访问的时间最长）的页。

首先在访问一个页的时候检测是否在链表中，如果在链表中则移动到链表头的位置：

```
static void
chect_list(uintptr_t addr) {  // 检查当前页是否在链表中，如果在就放在链表头
    list_entry_t *head = &pra_list_head, *le = head;

    while ((le = list_prev(le)) != head) {
        struct Page *curr = le2page(le, pra_page_link);
        if (curr->pra_vaddr == addr) {
            list_del(le);          // 删除找到的页
            list_add(head, le);    // 将页移到链表头部
            return;
        }
    }
}
```

如果要访问的页不在链表中，则删除链表尾的页并将要访问的页放在链表头的位置：

```
static int
_lru_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);

    // 选择最久未使用的页面进行替换
    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;
}
```

测试

测试函数如下：

```
static int
_lru_check_swap(void) {
    uintptr_t test_addrs[] = {0x3000, 0x1000, 0x4000, 0x2000, 0x5000};//vma里面没有0x6000
    unsigned char values[] = {0x0c, 0x0a, 0x0d, 0x0b, 0x0e};

    for (int i = 0; i < 5; i++) {
        write_and_check(test_addrs[i], values[i]);
        printlist();
    }
    write_and_check(0x1000, 0x0a);
    printlist();

    return 0;
}
```

测试结果如下：

```
zzr@zzr-virtual-machine:~/OS/riscv64-ucore-labcodes/lab3$ make qemu
+ cc kern/mm/swap.c
In file included from kern/mm/pmm.h:4,
                 from kern/mm/swap.h:6,
                 from kern/mm/swap.c:1:
kern/mm/swap.c: In function 'check_swap':
kern/mm/swap.c:217:19: warning: comparison of integer expressions of different signedness: 'int' and 'size_t' {aka 'long long unsigned int'} [-Wsign-compare]
  217 |      assert(total == nr_free_pages());
      |                   ^~
kern/debug/assert.h:17:15: note: in definition of macro 'assert'
   17 |         if (!(x)) {                                     \
      |               ^
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1
```

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc0200032 (virtual)
  etext  0xc02042a0 (virtual)
  edata  0xc020a040 (virtual)
  end    0xc0211564 (virtual)
Kernel executable memory footprint: 70KB
memory management: default_pmm_manager
membegin 80200000 memend 88000000 mem_size 7e00000
physcial memory map:
  memory: 0x07e00000, [0x80200000, 0x87ffffff].
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_vma_struct() succeeded!
Store/AMO page fault
page fault at 0x00000100: K/W
check_pgfault() succeeded!
check_vmm() succeeded.
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 0x3000 in lru_check_swap
--------list-head----------
vaddr: 3000
vaddr: 4000
vaddr: 2000
vaddr: 1000
--------list-end----------
write Virt Page 0x1000 in lru_check_swap
--------list-head----------
vaddr: 1000
vaddr: 3000
vaddr: 4000
vaddr: 2000
--------list-end----------
write Virt Page 0x4000 in lru_check_swap
--------list-head----------
vaddr: 4000
vaddr: 1000
vaddr: 3000
```

```
vaddr: 2000
--------list-end----------
write Virt Page 0x2000 in lru_check_swap
--------list-head----------
vaddr: 2000
vaddr: 4000
vaddr: 1000
vaddr: 3000
--------list-end----------
write Virt Page 0x5000 in lru_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
--------list-head----------
vaddr: 5000
vaddr: 2000
vaddr: 4000
vaddr: 1000
--------list-end----------
write Virt Page 0x1000 in lru_check_swap
--------list-head----------
vaddr: 1000
vaddr: 5000
vaddr: 2000
vaddr: 4000
--------list-end----------
count is 1, total is 8
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
```

`lru`策略成功实现

实验结果

`make grade`输出如下：

```
try to run qemu
qemu pid=8613
<<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check pmm:                                 OK
  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
Total Score: 45/45
```

实验中的知识点

内存管理与页表

在计算机中，内存管理是操作系统的一项重要任务，其中**页表**（Page Table）是实现虚拟内存的关键组件。虚拟内存允许程序使用比实际物理内存更大的地址空间，页表用于虚拟地址和物理地址之间的映射。以下是关于页表和分页的一些关键知识点：

1. 分页（Paging）

分页是内存管理的一种技术，它将内存划分为固定大小的块，每个块叫做"页"（Page）。虚拟内存和物理内存也都被划分为相同大小的页。

- **虚拟地址空间**：程序运行时，操作系统为其分配虚拟内存地址。
- **物理内存**：计算机的实际内存，包含了操作系统和程序使用的所有数据。

分页的目标是使得程序可以在虚拟地址空间中按页访问数据，而操作系统通过页表来将虚拟地址转换为物理地址。

2. 页表（Page Table）

页表是存储虚拟页与物理页之间映射关系的数据结构。每个虚拟页在页表中都有一个条目，包含其对应的物理页框的地址。

- **虚拟页号（VPN）**：表示虚拟内存中的一个页。
- **物理页框号（PFN）**：表示物理内存中的一个页框（物理内存的块）。

页表条目一般包括以下信息：

- **物理页框号**：映射到物理内存的地址。
- **有效位（Valid bit）**：指示该页是否在内存中有效。
- **访问权限位**：控制对该页的访问权限，例如可读、可写等。
- **脏位（Dirty bit）**：指示该页是否被修改过，需要写回磁盘。

3. 缺页异常（Page Fault）

当程序访问的虚拟地址对应的物理页不在内存中时，操作系统会触发**缺页异常**。缺页异常处理过程如下：

- 操作系统检查页表，发现虚拟地址没有映射到有效的物理地址。
- 操作系统将相应的页从磁盘（或其他外部存储设备）加载到内存中。
- 操作系统更新页表，并重新执行程序。

4. 多级页表（Multilevel Page Table）

在较大的地址空间中，单级页表可能会占用大量内存。为此，现代操作系统通常使用**多级页表**来减少内存消耗。

- **二级页表**：通过二级索引来减少页表的大小。第一级页表存储一个指向第二级页表的指针，第二级页表存储虚拟地址到物理地址的映射。
- **三级页表**：在某些系统中，使用三级或更多级的页表来进一步优化内存的使用。

5. 虚拟内存的优势

虚拟内存和页表的使用提供了以下几个主要优势：

- **隔离性**：每个进程都有独立的虚拟地址空间，防止了进程之间的内存干扰。
- **简化内存管理**：程序可以使用比物理内存更大的地址空间，操作系统通过分页机制管理物理内存。
- **内存共享与保护**：通过页表，可以实现进程间共享内存区域，同时控制进程对内存的访问权限。

页面置换策略

在虚拟内存系统中，**页面置换**是指当物理内存满时，操作系统需要决定哪些页面应该从内存中移除，以便腾出空间加载新的页面。不同的页面置换策略根据不同的算法选择移除页面的顺序。以下是三种常见的页面置换策略：**FIFO**（先进先出）、**Clock**、**LRU**（最近最少使用）。

1. FIFO（先进先出）

FIFO（First In First Out）是一种简单的页面置换策略，基于页面进入内存的顺序进行替换。

- **原理**：FIFO算法将最先进入内存的页面替换掉，类似于排队的方式，页面按进入内存的顺序排队，最先进入的页面会被最先淘汰。
- **优点**：实现简单，易于理解。
- **缺点**：不考虑页面的实际使用情况，可能会导致频繁使用的页面被提前替换，从而出现**Belady's Anomaly**（贝拉迪异常），即随着内存页数的增加，缺页率反而增加。

FIFO的工作过程：

1. 当一个新的页面需要加载到内存时，检查是否存在空闲的页面框。
2. 如果没有空闲框架，移除最早加载的页面，腾出空间。
3. 将新页面加载到被淘汰页面的位置。

2. Clock（时钟算法）

Clock算法是一种高效的页面置换算法，常被视为FIFO的改进版本。

- **原理**：Clock算法通过维护一个指向页面的指针，模拟时钟的走动来替换页面。当需要替换页面时，算法会检查指针指向的页面是否被访问过。若未被访问，替换该页面；若被访问，则将该页面的访问位清零，并将指针指向下一个页面。
- **优点**：比FIFO更加高效，且避免了贝拉迪异常。
- **缺点**：实现稍复杂，需要额外的访问位和指针。

Clock的工作过程：

1. 每个页面有一个访问位，初始值为0。
2. 当一个页面被访问时，将其访问位设为1。
3. 当需要替换页面时，从当前指针指向的页面开始检查，若访问位为0，则替换该页面；若访问位为1，则将其清零，并将指针指向下一个页面。

3. LRU（最近最少使用）

LRU（Least Recently Used）是一种根据页面的使用情况来决定替换策略的算法，优先替换那些最久没有被使用的页面。

- **原理**：LRU算法假设最久未使用的页面在未来也不太可能被使用，因此将其淘汰。实现LRU通常需要一个数据结构（如链表或栈）来追踪页面的访问顺序。
- **优点**：通过考虑页面的使用频率，LRU能够有效地减少缺页中断，提高内存使用效率。
- **缺点**：实现较复杂，尤其在硬件不支持直接追踪最近访问的页面时，可能需要额外的时间和空间开销。

LRU的工作过程：

1. 使用一个数据结构（如链表、栈或计数器）来跟踪页面的访问顺序。
2. 当一个页面被访问时，将该页面移动到数据结构的顶部（或最前面）。
3. 当需要替换页面时，选择最底部（或最后）的页面进行替换。

4. 总结

- **FIFO**：简单易实现，但可能导致贝拉迪异常。
- **Clock**：FIFO的优化版本，避免了贝拉迪异常，效率更高。
- **LRU**：基于页面的实际使用情况，通常能提供更好的性能，但实现相对复杂。

每种策略都有其优缺点，具体使用时需要根据系统的特点和需求进行选择。