

# 操作系统 lab3 实验报告

---

小组成员：

- 2110408 吴振华
  - 2112426 怀硕
  - 2113635 王祎宁
- 

做完实验二后，大家可以了解并掌握物理内存管理中页表的建立过程以及页面分配算法的具体实现。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，学习如何在磁盘上缓存内存页，从而能够支持虚拟内存管理，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。

---

## 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。

---

## 实验报告要求

---

对实验报告的要求：

- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

---

## 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？

**FIFO (First-In, First-Out)** 页面置换算法是一种最简单的页面置换策略。算法使用一个队列来维护当前内存中的页面，当一个新页面需要被装入内存时，它被添加到队列的末尾。当需要置换页面时，队列的最前面的页面（即最早被装入内存的页面）被选中。这种策略遵循“先进先出”的原则，即最早进入内存的页面最早被替换出去。

具体来说，在FIFO页面置换算法中，当一个页面从被换入到被换出的过程中，以下函数和宏被调用：

1. `swap_init(void)` :
  - 调用 `swapfs_init()` 初始化交换文件系统。
  - 初始化FIFO页面置换算法，设置 `swap_manager_fifo` 的各个成员函数指针。
  - 调用 `sm->init()` 初始化FIFO算法的内部数据结构。
2. `swap_init_mm(struct mm_struct *mm)` :
  - 调用 `_fifo_init_mm(mm)` 初始化FIFO算法的mm结构。
3. `swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)` :
  - 调用 `_fifo_map_swappable(mm, addr, page, swap_in)` 将页面加入FIFO队列，表示该页面最近被访问过。
4. `swap_out(struct mm_struct *mm, int n, int in_tick)` :
  - 多次调用 `_fifo_swap_out_victim(mm, &page, in_tick)`，从FIFO队列的最前端选取最早被访问的页面，将其从队列中移除，置换出去。
5. `swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)` :

- 调用 `swapfs_read()` 将页面从交换文件中读取到内存。
  - 分配一个新的页面，将读取的数据存入其中。
6. `_fifo_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)` :
- 从FIFO队列的最前端选取最早被访问的页面，将其从队列中移除，并将该页面的地址赋给 `ptr_page`，用于后续置换出。
7. `get_pte(mm->pgdir, addr, 0)` :
- 获取页面虚拟地址对应的页表项指针。
8. `list_init(&pra_list_head)` :
- 初始化FIFO算法使用的队列 `pra_list_head`。
9. `list_add(head, entry)` :
- 将页面链接到FIFO队列的末尾，表示该页面最近被访问。
10. `list_prev(head)` :
- 获取队列的最前端，即最早被访问的页面。
11. `list_del(entry)` :
- 将页面从FIFO队列中移除。
12. `le2page(entry, pra_page_link)` :
- 将`list_entry`转换为`Page`结构体，用于获取页面信息。
13. `swapfs_read()` :
- 从交换文件中读取页面数据。
14. `pte2page(*check_ptep[i])` :
- 将页表项转换为页面结构体，用于获取页面信息。
15. `assert()` :
- 在代码中进行断言检查，如果条件不成立则会触发panic，终止程序执行。在这里用于检查条件是否符合预期，如果不符合，则会触发panic，表示算法实现错误。

这些函数和宏的调用顺序遵循FIFO页面置换算法的逻辑，保证了最早被访问的页面最先被置换出去。

## 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。

### Question 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。

在上述提到的 `get_pte` 函数中，两段相似的代码用于处理页表的两级映射结构。它们负责确保给定虚拟地址 `la` 所在的页表条目存在于页表中，如果不存在则根据需要进行创建，并将相应的页表项设置为正确的物理地址和标志位（比如 `PTE_U` 和 `PTE_V`）。

这种结构是通用的，不论是在sv32、sv39还是sv48的RISC-V地址转换模型下，页表的组织方式都类似，因此这两段代码在不同的页表结构下可以共用。

以下是这两段代码的详细解释：

#### CodeBlock1

```
pde_t *pdep1 = &pgdir[PDX1(la)];
if (!(*pdep1 & PTE_V)) {
    // 如果顶级页表项不存在
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {
        // 如果不需要创建或者无法分配新页
        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);
}
```

- `PDX1(la)` 用于获取虚拟地址的顶级页表索引。
- 如果顶级页表项不存在（即未被标记为有效），则分配一个新的物理页，并将相应的页表项初始化为指向这个物理页的地址。
- `pte_create()` 函数用于创建一个页表项，将其标记为用户可访问（`PTE_U`）和有效（`PTE_V`）。
- 如果需要创建新页并且成功创建，将返回指向这个新页表项的指针。

## CodeBlock2

```
pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(1a)];
if (!(*pdep0 & PTE_V)) {
    // 如果次级页表项不存在
    struct Page *page;
    if (!create || (page = alloc_page()) == NULL) {
        // 如果不需要创建或者无法分配新页
        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);
}
```

- `PDX0(1a)` 用于获取虚拟地址的次级页表索引。
- `PDE_ADDR(*pdep1)` 用于获取顶级页表项所指向的次级页表的物理地址。
- 如果次级页表项不存在（即未被标记为有效），则分配一个新的物理页，并将相应的页表项初始化为指向这个物理页的地址。
- 如果需要创建新页并且成功创建，将返回指向这个新页表项的指针。

这两段代码一起保证了在两级页表结构下，当需要处理一个虚拟地址时，会按照顶级页表和次级页表的索引关系，检查相应的页表项是否存在，如果不存在则根据需要创建新的页表项，并且确保这些页表项指向了正确的物理页，并且设置了必要的标志位。这样，就能够确保虚拟地址被正确映射到物理地址。

## Question 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一个函数里是一种常见的做法，尤其是在实现简单的页表管理函数时。这种做法使得代码结构相对简单，尤其是对于只包含两级或者三级页表的体系结构。

但在实际的操作系统内核或者虚拟内存管理模块中，通常需要更加复杂的页表管理功能。例如，可能需要支持**页表项的缓存**、**惰性分配**、**写时复制**等高级特性。在这种情况下，将页表项的查找和分配拆分成独立的函数可能更为合适。

拆分这两个功能的主要优势包括：

- **模块化设计**：拆分功能可以使得代码更具模块性，方便单独测试和维护每个功能模块。
- **可重用性**：独立的分配函数可以被其他需要分配页表项的地方复用，提高了代码的可重用性。

- **可定制性：**不同的场景可能需要不同的分配策略，将分配功能拆分开来可以更容易地定制适应不同需求的算法或策略。
- **易于维护：**当需求变化时，可以更容易地修改或者替换其中一个功能，而不影响另一个功能。

综上所述，是否需要拆分 `get_pte` 函数取决于项目的需求和复杂性。在简单的场景下，将两个功能合并在一个函数可能是足够的。但是，在复杂的系统中，拆分这两个功能通常是更好的做法，以提高代码的可维护性和扩展性。

---

## 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。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

`do_pgfault` 用于处理页面错误，即在页面错误发生时，将缺失的页面加载到内存中，并更新页表，使得程序可以继续执行。

在给定的待补充函数实现中（Before */LAB3 EXERCISE 3: YOUR CODE/*），具体思路阐述如下：

- **获取当前进程的VMA（Virtual Memory Area）结构：**通过 `find_vma` 函数，找到包含发生页面错误的地址的VMA结构。
- **检查VMA的权限：**根据VMA的权限信息，判断页面错误的类型（读/写）。如果页面错误发生在不允许写入的区域，或者VMA不存在，函数将返回错误。
- **获取或创建PTE：**根据给定的地址，在页表中查找相应的PTE。如果PTE不存在，函数将根据地址和权限信息创建一个新的PTE，并将其映射到相应的物理页。

根据注释中的思路提示补充后的(部分)完整代码如下：

```

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*/
    if (swap_init_ok) {
        struct Page *page = NULL;
        // 将addr线性地址对应的物理页数据从磁盘交换到物理内存中(令Page指针指向交换成功
        后的物理页)
        if ((ret = swap_in(mm, addr, &page)) != 0)
        {
            cprintf("swap_in in do_pgfault failed\n");
            goto failed;
        }
        page_insert(mm->pgdir, page, addr, perm); //更新页表, 插入新的页表项
        swap_map_swappable(mm, addr, page, 1); //标记这个页面将来是可以再换出的
        page->pra_vaddr = addr;
    }
    else
    {
        cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
        goto failed;
    }
}

```

上述补充部分的具体思路阐述如下：

- **交换页面进入内存：**在这一部分，需要根据给定的地址，将相应的磁盘页面交换到内存中。使用swap\_in函数来完成这个任务，它会将磁盘上的数据读取到一个物理页中。
- **更新页表：**将物理页与线性地址进行映射，即将PTE中的物理地址与线性地址相关联。这样，程序就可以通过线性地址访问到相应的物理页。
- **使页面可交换：**调用swap\_map\_swappable函数，将页面标记为可交换的。这样，当系统需要释放内存时，可以将这些页面交换出去，而不是直接丢弃。

## Question 1

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。

页目录项（Page Directory Entry, PDE）和页表项（Page Table Entry, PTE）是用于构建分页机制的数据结构。它们在操作系统的内存管理中起到关键作用，同时也对页替换算法的实现有一定的影响，具体的 **潜在用处** 分析如下：



## PDE

- **多级页表管理**：UCore可以利用多级页表机制，将4GB的线性地址空间映射到物理内存上。通过多级页表，UCore可以更灵活地管理大内存空间。
- **内存隔离**：通过设置不同的PDE，UCore可以实现内存隔离，确保不同的进程不能直接访问其他进程的地址空间。
- **虚拟内存映射**：UCore可以通过修改PDE的内容，实现虚拟内存到物理内存的映射，包括内核虚拟地址到物理地址的映射。

## PTE

- **页面替换算法的支持**：PTE中的控制信息可以被页面替换算法用来辅助决策。例如，可以使用脏位（Dirty Bit）来判断页面是否被修改过，从而选择最适合替换的页面。
- **页面访问权限的控制**：PTE中的权限信息（读、写、执行权限）可以用于控制页面的访问权限，确保只有合法的操作可以访问页面内容。
- **页面状态标志**：PTE中的部分位可以用于表示页面的状态，例如是否被交换出去、是否被锁定等。这些状态信息可以用于操作系统的内存管理策略。

在UCore中，合理使用和管理这些页目录项和页表项的信息，可以帮助实现高效的内存管理和页面替换策略。通过合理地设置这些项，UCore可以在有限的物理内存资源下，实现更好的性能和内存利用率。

## Question 2.1

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？

当UCore的缺页服务例程在执行过程中访问内存并出现了页访问异常时，硬件会执行以下步骤：

1. **保存当前上下文**：硬件会自动保存当前进程的上下文，包括程序计数器（PC）和其他相关寄存器的值，将这些信息保存在内核栈中或者特定的异常堆栈中。
2. **触发异常处理**：硬件检测到页访问异常后，会引发异常，将控制权转移到操作系统内核的页访问异常处理例程。
3. **切换特权级别**：硬件会将处理器的特权级别从用户模式切换到内核模式，以便访问操作系统的数据结构和指令。
4. **提供异常信息**：硬件会将引发异常的原因和相关信息（如错误码）传递给操作系统，帮助操作系统确定异常的具体类型和原因。



5. **暂停当前指令的执行**：引发异常的指令的执行会被暂停，不会继续执行下去。

总的来说，硬件负责**检测并通知操作系统有关异常的发生**；操作系统则负责根据异常的类型采取适当的措施，例如加载缺失的页面数据、更新页表、并最终恢复被中断的程序的执行。

## Question 2.2

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？

在ucore中，`struct Page *pages` 这个全局变量表示了系统中所有的物理页，它是一个指向 `struct Page` 结构体数组的指针。这个数组的每一项代表一个物理页，包含了物理页的相关信息，比如引用计数、状态标志等。与之对应，PTE 和 PDE 则用来建立虚拟地址与物理地址的映射关系。它们之间的对应关系主要解析如下：

- **PDE 对应到 `struct Page` 的元素：**

- 在ucore中，页目录项用于建立虚拟地址到页表的映射关系。
- `struct Page *pages` 数组的每一项可以看作是一个页表，而页表的物理地址就是该 `struct Page` 元素的物理地址。
- 因此，PDE中的内容（指向页表的物理地址）可以对应到 `struct Page *pages` 数组的一个元素。

- **PTE 对应到 `struct Page` 的元素：**

- 页表项用于建立虚拟地址到物理页框地址的映射关系。
- 当一个虚拟地址被映射到物理页框时，该PTE中存储了物理页框的地址。
- `struct Page *pages` 数组中的每一项代表一个物理页，可以将PTE中的内容（指向物理页框的地址）与 `struct Page *pages` 数组中的一个元素对应起来。

---

## 练习4：补充完成Clock页替换算法（需要编程）

### 设计实现过程

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。

CLOCK 是一种页替换算法，用于操作系统中的虚拟内存管理。它是一种近似于最佳页面置换算法的改进算法。CLOCK算法使用一个环形缓冲区（类似于钟表的环形结构）来组织物理内存中的页

面。

算法的基本思想如下：

- **环形缓冲区**：CLOCK算法维护一个环形缓冲区，其中存储了物理内存中的页面。这个缓冲区类似于一个时钟，指针按照顺时针方向移动。
- **访问位**：对每一页维护一个访问位（也称为引用位或使用位）。访问位表示该页是否被访问过。当页面被访问时，访问位被置为1。
- **替换策略**：当需要替换页面时，CLOCK算法检查当前指针指向的页面的访问位。如果访问位为1，表示该页面被访问过，那么将访问位清零，然后继续移动指针。如果访问位为0，表示该页面未被访问过，那么选择该页面进行替换，并更新指针位置。
- **循环移动指针**：指针按照顺时针方向循环地遍历缓冲区。这样，CLOCK算法保证了每个页面都有机会被替换，同时也确保了被频繁访问的页面不容易被替换出去

基于以上思路，页替换算法实现的流程如下：

- 修改 `swap.c` 文件，首先本练习的本质要求仍是用 *Clock* 算法 替代给定的 *FIFO* 算法 并 `make qemu && make grade` 成功，故首先类比到之前 `pmm_manager` 的替换，将 `swap_manager` 进行替换，对 `swap.c/swap_init` 函数修改如下：

```
int swap_init(void){
    ...
    sm = &swap_manager_clock;    //use Clock Replacement Algorithm
    //sm = &swap_manager_fifo;    //use FIFO Replacement Algorithm
    ...
}
```

- 填充完善 `mm/swap_clock.c`，完成上述整体算法的替换后即可着手 *Clock* 算法 具体细节的填充实现，具体阐述如下：
  - `_clock_init_mm`，该函数是为了初始化算法所需的数据结构，包括一个链表头 `pra_list_head` 和两个指针 `curr_ptr` 和 `mm->sm_priv`。这些数据结构会在页面替换过程中被使用。具体实现注释已够详细，无需多言。

```
static int
_clock_init_mm(struct mm_struct *mm)
{
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // 初始化pra_list_head为空链表
    // 初始化当前指针curr_ptr指向pra_list_head，表示当前页面替换位置为链表头
    // 将mm的私有成员指针指向pra_list_head，用于后续的页面替换算法操作s
    list_init(&pra_list_head);
    curr_ptr = mm->sm_priv = &pra_list_head;
    return 0;
}
```

- **\_clock\_map\_swappable**，该函数的主要作用是将新访问的页面加入到替换队列中，并更新 curr\_ptr 指针，使其指向最新加入的页面。这样，当需要进行页面替换时，就可以从 curr\_ptr 指向的页面开始查找。这是CLOCK页面替换算法的基本思想，它通过维护一个指针 curr\_ptr，始终指向最老的、未被访问的页面，从而实现页面替换。

```
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);

    /*LAB3 EXERCISE 4: YOUR CODE*/
    // 将页面page插入到页面链表pra_list_head的末尾
    list_add(&(pra_list_head), entry);
    // 更新curr_ptr，使其指向新加入的页面
    curr_ptr = list_prev(&(pra_list_head));
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited = 1;

    return 0;
}
```

- list\_entry\_t \*entry = &(page->pra\_page\_link);：获取指向页面 page 的 pra\_page\_link 成员的指针。 pra\_page\_link 是一个ucore中用于页面替换算法的链表节点。
- assert(entry != NULL && curr\_ptr != NULL);：进行断言检查，确保 entry 和 curr\_ptr 不为空，如果为空，则触发断言失败。
- list\_add(&(pra\_list\_head), entry);：将页面 page 插入到页面链表 pra\_list\_head 的末尾。这个操作表示将最新访问的页面加入到替换队列中。

- `curr_ptr = list_prev(&(pra_list_head));` : 更新 `curr_ptr` 的位置, 使其指向新加入的页面。 `curr_ptr` 是一个静态指针, 它指向替换队列中当前指向的页面。
- `page->visited = 1;` : 将页面的 `visited` 标志置为1, 表示该页面已被访问。
- `_clock_swap_out_victim`, 该函数实现了CLOCK页面替换算法中选择牺牲页面 (victim page) 的逻辑, 即通过遍历页面链表 `pra_list_head`, 找到最早未被访问的页面, 作为牺牲页面返回。如果所有页面都已被访问过, 函数将循环遍历链表直到找到合适的牺牲页面。。

```
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) {
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 遍历页面链表pra_list_head, 查找最早未被访问的页面
        list_entry_t *le = curr_ptr;
        curr_ptr = list_next(curr_ptr);
        // 如果当前位置已经到达链表尾部, 将curr_ptr重新指向链表头部
        if (curr_ptr == head) {
            curr_ptr = list_next(head);
        }
        // 获取当前页面对应的Page结构指针
        struct Page *page = le2page(le, pra_page_link);
        // 检查visited位, 如果为0, 表示页面未被访问, 选择该页面作为牺牲页面
        if (page->visited == 0) {
            page->visited = 1;
            list_del(&(page->pra_page_link));
            // 将该页面指针赋值给ptr_page作为换出页面
            *ptr_page = page;
            return 0;
        } else {
            // 如果页面已被访问, 将visited标志置为0, 表示该页面已被重新访问
            page->visited = 0;
        }
    }
    return 0;
}
```

- `list_entry_t *le = curr_ptr;` : 获取当前指针 `curr_ptr` 指向的链表节点。
- `curr_ptr = list_next(curr_ptr);` : 将当前指针移动到下一个链表节点。

- `if (curr_ptr == head)`：检查当前指针是否已经到达链表尾部，如果是，将 `curr_ptr` 重新指向链表头部，形成一个循环链表。
- `struct Page *page = le2page(le, pra_page_link);`：将链表节点转换为对应的 `struct Page` 结构体指针，这里使用了宏 `le2page`，该宏的作用是根据链表节点的成员地址(`pra_page_link`)得到 `struct Page` 结构体的指针。
- `if (page->visited == 0)`：检查页面的 `visited` 标志，如果为0，表示页面未被访问。
- `page->visited = 1;`：将页面的 `visited` 标志置为1，表示该页面已被访问。
- `list_del(&(page->pra_page_link));`：从链表中删除该页面。
- `*ptr_page = page;`：将该页面的指针赋值给 `ptr_page`，作为选择的牺牲页面。
- 如果页面已被访问（`else` 部分），将 `visited` 标志置为0，表示该页面已被重新访问。

至此，已基本完成了 *Clock* 算法的实现，但经验证发现，要通过 `make grade` 的 `check`，还需在 `vmm.c/do_pgfault` 函数中加入对全局变量 `curr_ptr` 的指定格式输出，即在该函数体中（开头处）加入以下语句：

```
// cprint curr_ptr
extern list_entry_t *curr_ptr;
cprintf("curr_ptr %p\n", (void*)curr_ptr);
```

然后就可以进行 `make grade` 的验证了，效果如下：

观察上图可知，已成功实现了 *Clock* 算法 并通过了测试。

## Question

请回答如下问题：比较Clock页替换算法和FIFO算法的不同。

两种算法的主要区别阐述如下：

### CLOCK

- 选择逻辑：

- CLOCK算法维护一个环形链表（也可以看作是一个时钟），其中的页面按照访问位（也称为“使用位”或“R位”）被组织起来。当需要替换页面时，CLOCK算法从当前指针位置开

始，查找第一个访问位为0的页面，将其作为牺牲页面。如果所有页面的访问位都为1，则CLOCK算法将继续循环查找，直到找到适合替换的页面。

- **性能特点：**

- CLOCK算法相对于FIFO算法更加智能，因为它能够根据页面的访问情况进行选择，避免了FIFO算法的局限性。

## FIFO

- **选择逻辑：**

- FIFO算法维护一个队列，当需要替换页面时，选择队列中最早进入的页面，即先进入队列的页面。这种选择逻辑保证了最早被放入内存的页面会被最先替换出去。

- **性能特点：**

- FIFO算法非常简单，但是它有一个主要的缺点，即它不考虑页面的访问情况。即使一个页面在内存中频繁被访问，只要它是最早进入内存的，也会被替换掉。

从对比分析的角度阐述如下：

- **智能性：** CLOCK算法相对于FIFO算法更加智能，因为它考虑了页面的访问情况，选择时会尽量保留经常被访问的页面。
- **复杂性：** CLOCK算法相对于FIFO算法稍微复杂一些，因为它需要维护一个环形链表，并且需要检查页面的访问位。
- **公平性：** FIFO算法对所有页面的处理是公平的，即按照进入内存的顺序选择替换页面。而CLOCK算法在某些情况下可能更加“偏袒”那些被频繁访问但是刚刚被置为未访问的页面。

总的来说，CLOCK算法相对于FIFO算法在性能上更有优势，因为它考虑了页面的访问情况。但是，CLOCK算法相对复杂一些，需要额外的维护和操作。

## 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？

在操作系统中，使用“一个大页”的页表映射方式相对于分级页表的优劣势总结如下：

### 好处和优势

- **减少页表项数量：** 大页表映射减少了页表中的项数，因为一个大页可以映射更多的物理内存。这样可以减小页表的大小，提高查找速度和操作效率。

- **提高内存访问速度：** 大页表映射减少了页表的深度，因此在访问内存时需要更少的查找步骤。这可以提高内存访问速度，减少了内存访问的开销。
- **增加TLB (Translation Lookaside Buffer) 命中率：** TLB是一个高速缓存，存储了虚拟地址到物理地址的映射关系。使用大页表映射可以增加TLB的命中率，减少了TLB缺失 (TLB miss) 的次数，提高了内存访问速度。

## 坏处和风险

- **内存浪费：** 如果内存分配不是大页的整数倍，可能会导致内存浪费。因为如果一个小块内存分配在大页的一部分上，那么整个大页都会被占用，无法充分利用。
- **不灵活：** 大页表映射是一种静态的映射方式，不够灵活。在某些情况下，系统可能需要更精细的内存管理，例如处理不同大小的内存块，这时使用大页可能不够合适。
- **TLB污染：** 如果大页表映射的范围内存在不同的权限或属性，可能会导致TLB污染。TLB污染是指TLB中的一个表项被用于多个不同的页表项，这可能导致不一致性和安全问题。

总而言之，大页表映射适合需要快速访问大块内存的情况，但同时 also 需要注意内存的合理利用和系统的灵活性。

---

## 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）

---

### LRU算法：

最久未使用 (least recently used, LRU) 算法：利用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。

### 朴素想法：

每次访问页的时候，将访问的页的visited值置零，表示刚刚访问过，其他的页的visited则加1，表示一定时间没有使用。

主要添加了 `_lru_accessed_check` 函数：



```

static int
_lru_accessed_check(struct mm_struct *mm)
{
    cprintf("\nbegin accessed check-----\n");
    list_entry_t *head = (list_entry_t *)mm->sm_priv;
    assert(head != NULL);
    list_entry_t *entry = head;
    while ((entry = list_prev(entry)) != head)
    {
        struct Page *entry_page = le2page(entry, pra_page_link);
        pte_t *tmp_pte = get_pte(mm->pgdir, entry_page->pra_vaddr, 0);
        cprintf("the ppn value of the pte of the vaddress is: 0x%x  \n", (*tmp_pte) >>
10);
        if (*get_pte(mm->pgdir, entry_page->pra_vaddr, 0) & PTE_A)
        {
            entry_page->visited = 0;
            *get_pte(mm->pgdir, entry_page->pra_vaddr, 0) = *get_pte(mm->pgdir,
entry_page->pra_vaddr, 0) ^ PTE_A;
        }
        else
        {
            entry_page->visited++;
        }
        cprintf("the visited goes to %d\n", entry_page->visited);
    }
    cprintf("end accessed check-----\n\n");
}

```

在每次换入/换出的时候，调用该函数。

实现方法：

遍历整个链表：

- 若访问位被置位：
  - 将该页的visited置0，表示刚刚访问。
  - 随即将访问位A置0，方便后续统计。
- 若访问位A位0：
  - 对应页的visited值加一，表示未被访问，最近时间++。

**选择换出对象函数：**

```

static int
_lru_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)
{
    _lru_accessed_check(mm);
    list_entry_t *head = (list_entry_t *)mm->sm_priv;
    assert(head != NULL);
    assert(in_tick == 0);
    // 先选第一个
    list_entry_t *entry = list_prev(head);
    uint_t largest_visted = le2page(entry, pra_page_link)->visited;
    curr_ptr = entry;
    while (1)
    {
        // 遍历找到最大的visited, 表示最早被访问的
        entry = list_prev(entry);
        if (entry == head)
        {
            break;
        }
        if (le2page(entry, pra_page_link)->visited > largest_visted)
        {
            largest_visted = le2page(entry, pra_page_link)->visited;
            le2page(entry, pra_page_link)->visited = 0;
            curr_ptr = entry;
            continue;
        }
    }
    list_del(curr_ptr);
    *ptr_page = le2page(curr_ptr, pra_page_link);
    cprintf("curr_ptr %p\n", curr_ptr);
    return 0;
}

```

通过遍历整个链表寻找visited值最大的项，即最久没有使用的，作为换出对象。