Lab3:缺页异常和页面置换

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？

解答：

```
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();//这里alloc_page()内部可能调用swap_out()
     //找到对应的一个物理页面
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);//找到/构建对应的页表项
    //将物理地址映射到虚拟地址是在swap_in()退出之后，调用page_insert()完成的
     int r;
     if ((r = swapfs_read((*ptep), result)) != 0)//将数据从硬盘读到内存
     {
        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;
}
```

`alloc_page()` 函数分配一个新的物理内存页，用于存储从磁盘加载的数据。该函数会在内部调用 `swap_out()` 来将当前内存中的某些页面交换到磁盘。

`get_pte(mm->pgdir, addr, 0)` 函数根据进程的页目录 `mm->pgdir` 和虚拟地址 `addr` 查找该地址的页表项。如果该页表项不存在，参数 `0` 表示不会创建新的页表项，而是只查找现有的。通过此页表项可以获得当前虚拟地址是否有映射以及它的权限。

`swapfs_read((\*ptep), result)` 函数将物理页面 `result` 加载数据，`(*ptep)` 作为页表项指向交换空间中的某个位置（通常包含交换页面的索引），然后将对应的磁盘内容读入该内存页面。返回值 `r` 表示读取操作是否成功。

`page_insert(mm->pgdir, page, addr, perm)` 函数会将物理内存页 `page` 与虚拟地址 `addr` 进行映射，并将权限 `perm` 应用到页表项中。这通常是在 `swap_in` 完成后，调用此函数将数据加载到内存后进行的操作。

`assert(result != NULL)` 该宏用于检查 `alloc_page()` 是否成功分配了页面，确保 `alloc_page()` 返回的物理页面指针 `result` 不为空。如果为空，程序会触发断言失败，表示页面分配失败。

```
int swap_out(struct mm_struct *mm, int n, int in_tick)
{
     int i;
     for (i = 0; i != n; ++ i)
     {
          uintptr_t v;
          struct Page *page;
```

```
            int r = sm->swap_out_victim(mm, &page, in_tick);//调用页面置换算法的接口
            //r=0表示成功找到了可以换出去的页面
           //要换出去的物理页面存在page里
            if (r != 0) {
                    cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                    break;
            }

            cprintf("SWAP: choose victim page 0x%08x\n", page);

            v=page->pra_vaddr;//可以获取物理页面对应的虚拟地址
            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) {
                        //尝试把要换出的物理页面写到硬盘上的交换区，返回值不为0说明失败了
                    cprintf("SWAP: failed to save\n");
                    sm->map_swappable(mm, v, page, 0);
                    continue;
            }
            else {
                //成功换出
                    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);
            }
            //由于页表改变了，需要刷新TLB
            //思考：  swap_in()的时候插入新的页表项之后在哪里刷新了TLB?
            tlb_invalidate(mm->pgdir, v);
        }
        return i;
}
```

`sm->swap_out_victim(mm, &page, in_tick)` 调用页面置换算法，选择一个"受害者"页面进行交换。选择一个可以换出的物理页面，并通过 `page` 参数返回该页面。如果算法返回 `r != 0`，表示没有找到可以换出的页面，函数会打印错误信息并终止操作。

`get_pte(mm->pgdir, v, 0)` 函数通过页目录 `mm->pgdir` 查找虚拟地址 `v` 的页表项。这里传入 `0` 表示不会创建新页表项，只是查找现有的。通过该页表项可以验证当前页面是否有效。

`assert((\*ptep & PTE_V) != 0)` 该宏检查该页表项的有效位是否被设置，确保页面是有效的。`PTE_V` 是页表项中的有效位，表示该虚拟地址是否已映射到物理内存。如果 `PTE_V` 位没有设置，则会触发断言错误。

`swapfs_write((page->pra_vaddr / PGSIZE + 1) << 8, page)` 将指定的物理页面内容写入磁盘上的交换空间。这里 `page->pra_vaddr / PGSIZE + 1` 计算出该页面的交换区条目（即页面的索引），`<< 8` 是为了适配交换区格式，传递给 `swapfs_write` 函数。

`sm->map_swappable(mm, v, page, 0)` 如果写入交换空间失败，调用此函数将页面标记为不可交换状态。这个步骤是在失败的情况下执行，确保不再尝试将该页面交换出去。

`free_page(page)` 如果页面成功写入交换空间，调用 `free_page` 释放页面，表示该页面不再需要分配给进程，已被交换到磁盘。

`tlb_invalidate(mm->pgdir, v)` 当页表项发生变化时，需要刷新 TLB。`tlb_invalidate` 用来使得 CPU 中缓存的页表项失效，确保下一次访问该虚拟地址时能重新加载正确的页表项。

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。
> - 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？

理解get_pte()函数

get_pte()函数是用于在给定的页目录 `pgdir` 和虚拟地址 `la` 的情况下，获取对应的页表项（`pte_t`）。如果 `create` 参数为真，并且在必要时会创建缺失的页目录项和页表项。

它接受三个参数：`pde_t *pgdir`：指向页目录的指针；`uintptr_t la`：虚拟地址；`bool create`：表示是否在需要时创建页表项和页目录项。

它大致的实现过程为：

1. 首先通过 `PDX1(la)` 获取虚拟地址 `la` 在页目录中的索引，然后取该索引对应的页目录项 `pdep1`。
   - 如果该页目录项的 `PTE_V`（Present 标志位）未设置，表示对应的一级页表不存在。
   - 如果 `create` 为假或者分配页失败（即 `alloc_page()` 返回 `NULL`），则返回 `NULL`。
   - 否则，分配一个新页，设置页的引用计数为 1，获取该页的物理地址 `pa`，将对应内核虚拟地址空间的内容清零，然后设置页目录项 `pdep1`，使其指向新分配的页，并设置相应的标志位（`PTE_U` 和 `PTE_V`）。

```
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);
}
```

2. 接着通过 `PDX0(la)` 获取虚拟地址在一级页表中的索引，然后取该索引对应的页目录项 `pdep0`。这里 `pdep0` 实际上是指向二级页表的页目录项。
   - 如果该页目录项的 `PTE_V` 未设置，表示对应的二级页表不存在。
   - 如果 `create` 为假或者分配页失败（即 `alloc_page()` 返回 `NULL`），则返回 `NULL`。
   - 否则，分配一个新页，设置页的引用计数为 1，获取该页的物理地址 `pa`，将对应内核虚拟地址空间的内容清零，然后设置页目录项 `pdep0`，使其指向新分配的页，并设置相应的标志位（`PTE_U` 和 `PTE_V`）。

```
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];
//    pde_t *pdep0 = &((pde_t *)(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);
 //     memset(pa, 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
```

3. 最后，通过 `PTX(la)` 获取虚拟地址在二级页表中的索引，返回对应的页表项指针。

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

sv32，sv39，sv48的异同

`SV32`、`SV39` 和 `SV48` 都是 `RISC-V` 架构中的不同虚拟内存系统实现，它们的主要异同点如下：

相同点

都是为了实现虚拟内存管理，将虚拟地址映射到物理地址，提供内存保护和多进程隔离等功能。

不同点

一、地址空间大小

- **SV32**：32 位虚拟地址空间，通常能提供 4GB 的虚拟地址空间。
- **SV39**：39 位虚拟地址空间，提供更大的虚拟地址空间，理论上可以达到 512GB。
- **SV48**：48 位虚拟地址空间，提供非常大的虚拟地址空间，可满足大规模内存需求和复杂系统的要求

二、页表结构和层次

- **SV32**：页表结构相对简单，可能层次较少。通常采用两级页表或简单的页表结构来实现地址转换。
- **SV39**：页表结构可能更为复杂，可能需要更多层次的页表来实现高效的地址转换，以适应较大的地址空间。
- **SV48**：由于地址空间更大，页表结构可能更加复杂，可能需要更多层次的页表以及更精细的管理机制来实现高效的地址转换。

两段代码相似的原因

sv39页表结构特点

`SV39` 通常采用多级页表结构来管理虚拟地址空间。一般来说，可能会有页目录、中间级页表和页表等层次。这两段代码分别对应不同层次的页表项处理。

处理逻辑相似

判断页表项存在性

两段代码首先都要判断当前层级的页表项是否存在。通常是通过检查特定的标志位来确定，比如检查页表项中的 `“Present”` 标志位。如果该标志位未设置，就表示对应的页表（或下一级页表）不存在。

处理页表项不存在的情况

如果页表项不存在且参数 `create` 允许创建新的页表项，那么就会进行一系列相似的操作。

- 首先调用 `alloc_page()` 分配一个新的物理页。
- 接着设置新分配页的引用计数，通常使用 `set_page_ref(page, 1)` 来表示该页被引用了一次。
- 然后获取新分配页的物理地址 `pa`，并使用 `memset(KADDR(pa), 0, PGSIZE)` 将该物理页对应的内核虚拟地址空间清零，以确保新页的初始状态是干净的。
- 最后设置当前层级的页表项，使其指向新分配的页，并设置合适的标志位，通常包括 `PTE_U`（User can access）和 `PTE_V`（Present）等标志位，以表示该页可被用户访问且当前存在于内存中。

综上，两段代码十分相似，因为它们都是在处理页表项的查找和创建过程。对于不同层级的页表项，基本的处理逻辑和操作步骤是一致的。这种相似性是由 `sv39` 的多级页表结构以及实现虚拟地址到物理地址映射的基本需求所决定的。

查找和分配功能

我认为将页表项的查找和页表项的分配合并在一个函数里这种写法并不好。虽然对于使用者来说，只需要调用一个函数就可以完成页表项的查找和分配操作，使用起来比较方便。减少了函数调用的层级，使代码更加简洁。

但是，查找和分配是两个不同的功能，合并在一个函数中会增加功能的耦合度。如果未来需要对查找或者分配的逻辑进行修改，或者说要增加一级页表，可能会影响到整个函数，增加了代码维护的难度。并且，如果其他部分的代码只需要查找页表项而不需要分配，或者只需要分配页表项而不需要查找，那么这个合并的函数就不太适用，降低了代码的可重用性。

可以将这两个功能拆开为两个单独的函数，或者将创建某一级页表的代码独立成一个模块，在查找中调用，从而增强代码的复用性。

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。

解答：

设计实现过程：

```
swap_in(mm, addr, &page);
```

`swap_in` 函数是一个从磁盘交换空间读取页面数据到内存的操作。它会根据虚拟地址 `addr` 以及虚拟内存结构 `mm` 处理页面交换。

函数会分配一个新的物理页面，并通过页表项（PTE）中的交换条目（指向磁盘位置）来加载相应的数据。`page` 是加载到内存中的物理页面指针。

```
page_insert(mm->pgdir, page, addr, perm);
```

将加载的物理页面 `page` 插入到进程的页表中，并建立虚拟地址 `addr` 与物理地址 `page` 的映射关系，同时设置该页面的访问权限 `perm`。

```
swap_map_swappable(mm, addr, page, 1);
```

`swap_map_swappable(mm, addr, page, 1)` 将页面标记为可交换，这意味着该页面在未来可以被交换出去，以便释放内存。

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。**

页替换算法的核心目标是决定在内存不足时，哪些页面应该被换出到磁盘，以便腾出内存空间。在此过程中，PDE 和 PTE 中存储的信息可以帮助操作系统做出决策。

页表项中的访问权限标志

页表项中的标志位对于选择哪一页被替换是至关重要的。例如，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。

PDE 和 PTE 中的物理页地址

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。

- **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?**

  此时会固定的跳转到初始化 `stvec` 时设置好的处理程序地址，也即 `alltraps` 处，进行上下文保存，以及将发生缺页中断的地址保存到 `trapframe` 中。然后跳转到中断处理函数 `trap()`，具体由 `do_pgfault` 处理，解决完毕返回到 `trapret` 恢复保存的寄存器，也即上下文，通过 `sret` 跳转回原程序。

  - **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的全局变量是一个用于管理物理内存页的数组，每个Page结构体记录了一个物理页的属性和状态。页表中的页目录项和页表项是用于实现虚拟地址到物理地址的映射关系的数据结构，每个页目录项或页表项记录了一个虚拟页对应的物理页的起始地址和一些标志位等信息。

    数据结构Page的全局变量与页表中的页目录项和页表项之间没有直接的对应关系，但是它们都涉及到物理内存页的管理和使用。 `PTE` 中存储的物理地址与 `Page` 结构体的物理页地址是相关的，`PDE` 通过指向页表间接影响 `Page` 结构体的管理。

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。数据结构Page的全局变量可以通过物理地址找到对应的Page结构体，而页表中的页目录项和页表项可以通过其高20位的虚拟地址找到对应的物理地址。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。

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）。(提示:要输出curr_ptr的值才能通过make grade)
>
> 请在实验报告中简要说明你的设计实现过程。请回答如下问题：
>
> - 比较Clock页替换算法和FIFO算法的不同。

补充了`mm/swap_clock.c`的`_clock_init_mm`、`_clock_map_swappable`和`_clock_swap_out_victim`函数

_clock_init_mm函数

- 初始化 `pra_list_head` 为空链表。
- 将 `curr_ptr` 指向 `pra_list_head`，表示初始时从链表头开始查找换出页面。
- 让传入的内存管理结构 `mm` 的私有成员指针 `sm_priv` 指向 `pra_list_head`，以便后续从 `mm_struct` 访问这个链表进行页面替换操作。

```
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: 2213906*/
     // 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
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     list_init(&pra_list_head);
     curr_ptr = &pra_list_head;
     mm->sm_priv = &pra_list_head;
     return 0;
}
```

_clock_map_swappable函数

- 将新的可交换页面 `page` 插入到链表中 `curr_ptr` 的前面，表示最近到达的页面放在这里。同时将该页面的 `visited` 标志置为 1，表示已被访问。

```
_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: 2213906*/
    // 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_add_before(curr_ptr, entry);
    page->visited = 1;
    return 0;
}
```

_clock_swap_out_victim函数

- 从链表中查找合适的换出页面。不断遍历链表，检查页面的 `visited` 标志。如果页面未被访问，则将其从链表中删除，并将该页面指针赋值给 `ptr_page` 作为换出页面，然后更新 `curr_ptr`。如果页面已被访问，则将 `visited` 标志置为 0，表示该页面已被重新访问。

```
_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: 2213906*/
        // 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);
        }
        struct Page *page = le2page(curr_ptr, pra_page_link);
        if(!page->visited){
           cprintf("curr_ptr %p\n",curr_ptr);
           list_del(curr_ptr);
           *ptr_page=page;
           curr_ptr = list_next(curr_ptr);
           break;
        }
        else{
            page->visited=0;
            curr_ptr = list_next(curr_ptr);
        }
    }
    return 0;
}
```

如图为make grade的结果

```
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=53512
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check pmm:                                   OK
  -check vmm:                                   OK
  -check swap page fault:                       OK
  -check ticks:                                 OK
Total Score: 45/45
```

成功通过测试

Clock页替换算法和FIFO算法的不同

一、数据结构和关键变量

链表结构使用方式

- **FIFO**：仅使用一个链表`pra_list_head`来管理所有可交换页面，新页面总是添加到链表尾部，进行页面替换时从链表头部选择最早进入的页面。
- **Clock**：除了链表`pra_list_head`外，还引入了一个指针`curr_ptr`，用于在链表中遍历以找到合适的换出页面。

二、页面插入策略

新页面插入位置

- **FIFO**：新页面总是添加到链表尾部，表示最近到达的页面放在链表末尾，严格按照时间顺序排队。
- **Clock**：新的可交换页面插入到链表中`curr_ptr`的前面，不是简单地添加到尾部，插入位置更加灵活。

三、页面选择策略

选择换出页面的依据

- **FIFO**：直接从链表头部选择最早到达的页面作为换出页面，不考虑页面的访问情况。
- **Clock**：遍历链表，检查页面的“访问标志”（`visited`）。如果页面未被访问，则将其换出；如果页面已被访问，则将访问标志重置为 0，表示该页面已被重新访问，继续遍历查找下一个未被访问的页面。

四、性能和适应性

对页面访问模式的适应性

- **FIFO**：性能较差，因为它完全不考虑页面的访问频率和最近使用情况，只按照页面进入的时间顺序进行替换，可能会频繁替换正在使用的页面。
- **Clock**：相对更智能，考虑了页面的访问情况，对于具有局部性的页面访问模式，能够减少不必要的页面替换，提高性能。

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?

好处和优势：

1. **减少页表项数量**：
   - “一个大页”方式映射较大范围的物理地址空间，因此不需要多级页表那样复杂的结构和多层次的表项，可以减少内存开销。
   - 在极端情况下，如果整个进程所需的虚拟地址空间可以用一个大页映射完成，那么页表本身可以非常小或不需要额外的页表。
2. **TLB（转换后备缓冲区）命中率**：
   - 使用大页会减少页表项的数量，意味着虚拟到物理地址的映射关系变少，可以更容易保存在TLB中，从而提高TLB命中率。
   - 高TLB命中率直接减少页表访问开销，加快地址转换速度，提升性能。
3. **简化页表管理**：
   - 没有多级页表层次结构的复杂性，内存管理机制的实现更简单。
   - 在特定系统或应用中，可以通过更简单的映射方式，减少页表管理代码的复杂度。
4. **适合特定应用场景**：
   - 在嵌入式系统、实时系统或其他地址空间需求相对固定的场景中，大页表可能更为高效，减少管理开销，能获得稳定的性能。

缺点和风险

1. **内存浪费（碎片化）**：
   - 使用大页时，如果进程只需要一个小范围的地址空间，但大页映射则会分配更大块的物理内存，可能造成大量浪费。
   - 对于多个小程序或碎片化的内存需求场景，大页映射导致大量不可利用的空间，影响整体内存效率。
2. **缺乏灵活性**：
   - 大页映射缺乏多级页表那样的精细控制。分级页表允许按页的粒度管理虚拟内存，可以更灵活地映射不同区域。
   - 对于需要频繁动态分配或更改映射关系的系统，大页映射方式不适合。
3. **降低页面置换的效率**：
   - 大页使得换出操作的粒度变大。例如，当需要换出一个页面时，大页映射会导致较大块的内存换出。
   - 这种大粒度的页面管理会导致页面置换效率低下，尤其在需要频繁进行页面换入换出时。
4. **安全性和隔离性降低**：
   - 分级页表提供多级分段保护，而大页映射方式可能导致更多的地址空间暴露，增加信息泄露的风险。
   - 多级页表可以为不同的内存段提供不同的权限控制，大页映射则不具备这样的精细权限管理，可能导致安全隔离变差。
5. **硬件和操作系统支持限制**：
   - 一些体系结构或操作系统对大页的支持不够友好，可能需要特殊配置或扩展支持。特别是在通用操作系统中，可能缺乏对大页的灵活支持。

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）

> challenge部分不是必做部分，不过在正确最后会酌情加分。需写出有详细的设计、分析和测试的实验报告。完成出色的可获得适当加分。

LRU（Least Recently Used，最近最少使用）页面替换算法是一种**基于页面最近访问时间**的页面替换策略。该算法假设最近访问的页面在短时间内会再次被访问，而最久未被访问的页面可能在未来也不再使用。因此，每当需要替换页面时，LRU算法选择最久未使用的页面进行替换。

LRU算法的工作原理

LRU算法的核心是通过记录页面的**最近访问时间**，选择最久未被访问的页面进行替换。其基本步骤如下：

1. **页面访问记录**：系统维护一个记录结构（如链表、栈、时间戳等），按页面的最近访问顺序排列。
2. **页面访问更新**：每次页面被访问后，将该页面标记为最近访问。不同的实现方式可能将页面移到链表末尾或更新时间戳，以便能找到下一个最久未使用的页面。
3. **页面置换**：当发生页面缺页错误，且页面缓冲区已满，需要替换一个页面时，选择访问记录中**最久未使用的页面**进行替换。

LRU算法的优缺点

优点：

- **命中率较高**：相比于先进先出（FIFO）等算法，LRU更符合程序的**局部性原理**，尤其在有较强时间局部性的场景下表现出色。
- **简单直观**：基于最近访问的策略逻辑清晰，便于理解和实现。

缺点：

- **开销较大**：为了记录访问顺序，通常需要额外的空间和处理时间。如果用链表实现，页面每次被访问时都要更新链表位置；用时间戳实现则需要大量的比较操作。
- **硬件支持较弱**：在现代硬件中，直接硬件支持LRU较少，多数需要通过软件模拟。

示例

假设页面序列为 `A，B，C，D，A，E，F，A，B，C，D，E`，页面缓冲区的大小为 3。按照 LRU 页面置换的过程如下：

1. **缓冲区**初始为空。
2. **访问 A**：A 加入缓冲区 `[A]`
3. **访问 B**：B 加入缓冲区 `[A，B]`
4. **访问 C**：C 加入缓冲区 `[A，B，C]`
5. **访问 D**：页面缓冲区满，替换最久未使用的 A，缓冲区变为 `[B，C，D]`
6. **访问 A**：替换最久未使用的 B，缓冲区变为 `[A，C，D]`
7. **访问 E**：替换最久未使用的 C，缓冲区变为 `[A，D，E]`
8. **访问 F**：替换最久未使用的 D，缓冲区变为 `[A，E，F]`
9. 以此类推...

代码实现

修改`do_pgfault`函数

`do_pgfault`是处理页面缺页的函数，当系统缺页的时候。会调用该函数，我们添加上`lru_pgfault`的调用来处理缺页，完整代码如下：

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    pte_t* temp = NULL;
    temp = get_pte(mm->pgdir, addr, 0);

    // Check if the page is already valid and readable
    if (temp != NULL && (*temp & (PTE_V | PTE_R))) {
        return lru_pgfault(mm, error_code, addr);
    }//直接调用 lru_pgfault 处理缺页，返回处理结果

    int ret = -E_INVAL;
    struct vma_struct *vma = find_vma(mm, addr);
    pgfault_num++;

    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and can not find it in vma\n", addr);
        goto failed;
    }

    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }

    // Remove read permission as specified
    perm &= ~PTE_R;

    addr = ROUNDDOWN(addr, PGSIZE);
    ret = -E_NO_MEM;
    pte_t *ptep = get_pte(mm->pgdir, addr, 1);

    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;
            swap_in(mm, addr, &page);
            page_insert(mm->pgdir, page, addr, perm);
            swap_map_swappable(mm, addr, page, 1);
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
    }
```

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

完善`lru_pgfault`缺页处理函数

我们打印完缺页信息后，设置所有页面为不可读

```
if(swap_init_ok)
    unable_page_read(mm);
```

如果交换（swap）功能已经初始化，将调用 `unable_page_read(mm)` 将所有页面设置为不可读。这是为了确保在需要时再设置页面为可读，以便在 LRU 机制中确定页面的访问顺序。

接下来再获取并设置页面的读权限：

```
pte_t* ptep = NULL;
ptep = get_pte(mm->pgdir, addr, 0);
*ptep |= PTE_R;
```

- `ptep` 是页表项的指针，用于操作页表项的权限位。
- 调用 `get_pte` 获取虚拟地址 `addr` 的页表项，并将其读权限位 `PTE_R` 设置为 1，即设置该页面为可读。

然后我们检查是否启用了交换功能：

```
if(!swap_init_ok)
    return 0;
```

如果交换功能未初始化，直接返回 0，因为不需要进一步的页面管理操作。

然后我们就获取页面并将其放在链表头：

```
struct Page* page = pte2page(*ptep);
list_entry_t *head=(list_entry_t*) mm->sm_priv, *le = head;
while ((le = list_prev(le)) != head)
{
    struct Page* curr = le2page(le, pra_page_link);
    if(page == curr) {
        list_del(le);
        list_add(head, le);
        break;
    }
}
```

- `pte2page`：将页表项 `*ptep` 转换为 `Page` 结构体的指针 `page`。
- `head` 是页面链表的头部指针，`mm->sm_priv` 用于存储 LRU 页面链表的头部。
- `le = list_prev(le)`：遍历链表，从尾部往头部找到页面 `page` 对应的节点 `le`。
- 如果找到当前页面 `page`，则将其从链表中删除，然后将其插入到链表头部，以表明这是最近访问过的页面。

最后返回结果。

接下来是一些基本的函数,我们简单看看:

初始化内存管理

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

- `list_init(&pra_list_head)` 初始化链表头 `pra_list_head`。
- 将 `mm->sm_priv` 指向 `pra_list_head`，将链表头存储在 `mm` 结构体中。

映射可交换的页面

```
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);
    list_add((list_entry_t*) mm->sm_priv,entry);
    return 0;
}
```

- 将新的页面添加到 LRU 链表的头部。
- `list_add` 将页面 `page` 插入到 `pra_list_head` 链表中，表示该页面刚刚被访问过。

替换一个页面

```
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);
    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;
}
```

- `list_prev(head)` 获取链表尾部的页面，尾部页面是 LRU 页面（最近最少使用）。
- 将链表尾部页面删除并返回，作为需要替换的页面。

测试：

我们主要来查看是否每次访问后都放在链表头，以及每次替换的内存是否是最久未使用的：

```
static int
_lru_check_swap(void) {
    print_mm_list();
    cprintf("write Virt Page c in lru_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    print_mm_list();
    cprintf("write Virt Page b in lru_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    print_mm_list();
    cprintf("write Virt Page e in lru_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    print_mm_list();
    return 0;
}
```

我们查看输出的结果：

```
--------begin----------
vaddr: 0x4000
vaddr: 0x3000
vaddr: 0x2000
vaddr: 0x1000
---------end-----------
write Virt Page c in lru_check_swap
Store/AMO page fault
page fault at 0x00003000: K/W
lru page fault at 0x3000
--------begin----------
vaddr: 0x3000
vaddr: 0x4000
vaddr: 0x2000
vaddr: 0x1000
---------end-----------
write Virt Page b in lru_check_swap
Store/AMO page fault
page fault at 0x00002000: K/W
lru page fault at 0x2000
--------begin----------
vaddr: 0x2000
vaddr: 0x3000
vaddr: 0x4000
vaddr: 0x1000
---------end-----------
write Virt Page e in lru_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
Store/AMO page fault
page fault at 0x00005000: K/W
lru page fault at 0x5000
--------begin----------
```

```
vaddr: 0x5000
vaddr: 0x2000
vaddr: 0x3000
vaddr: 0x4000
---------end----------
count is 1, total is 8
```

```
page fault at 0x00004000: K/W
lru page fault at 0x4000
set up init env for check_swap over!
--------begin----------
vaddr: 0x4000
vaddr: 0x3000
vaddr: 0x2000
vaddr: 0x1000
---------end----------
write Virt Page c in lru_check_swap
Store/AMO page fault
page fault at 0x00003000: K/W
lru page fault at 0x3000
--------begin----------
vaddr: 0x3000
vaddr: 0x4000
vaddr: 0x2000
vaddr: 0x1000
---------end----------
write Virt Page b in lru_check_swap
Store/AMO page fault
page fault at 0x00002000: K/W
lru page fault at 0x2000
--------begin----------
vaddr: 0x2000
vaddr: 0x3000
vaddr: 0x4000
vaddr: 0x1000
---------end----------
write Virt Page e in lru_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
Store/AMO page fault
page fault at 0x00005000: K/W
lru page fault at 0x5000
--------begin----------
vaddr: 0x5000
vaddr: 0x2000
vaddr: 0x3000
vaddr: 0x4000
---------end----------
count is 1, total is 8
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
wz@wz-virtual-machine:~/riscv64-ucore-labcodes/lab3$
```

本实验中的知识点

基本概念

虚拟内存：

- 虚拟内存是一种计算机系统内存管理技术，它使得程序可以使用比实际物理内存更大的地址空间。
- 程序使用虚拟地址进行内存访问，而操作系统将虚拟地址转换为物理地址，以便在物理内存中进行实际的数据存储和读取。

页表：

- 页表是一种数据结构，用于记录虚拟地址和物理地址之间的映射关系。
- 每个进程都有自己的页表，操作系统通过页表来实现虚拟内存到物理内存的转换。

多级页表：

- 当虚拟地址空间非常大时，使用单级页表会占用大量的内存空间。多级页表通过将页表分成多个层次，减少了页表占用的内存空间。
- `RISC-V` 架构中包括`SV32`、`SV39` 和 `SV48` 等不同虚拟内存系统实现

使用多级页表进行虚拟内存管理的步骤

虚拟地址划分：

- 将虚拟地址划分为多个部分，每个部分对应多级页表中的不同层次。
- 例如，对于一个 32 位的虚拟地址和二级页表结构，可以将虚拟地址划分为高 10 位作为一级页表索引，中间 10 位作为二级页表索引，低 12 位作为页内偏移量。

页表查找：

- 当程序进行内存访问时，操作系统首先根据虚拟地址的高位部分在一级页表中查找对应的页表项。
- 如果一级页表项中指示存在对应的二级页表，则根据虚拟地址的中间部分在二级页表中查找对应的页表项。
- 二级页表项中包含了虚拟页面到物理页面的映射关系，以及一些标志位，如存在位、读写位等。

物理地址计算：

- 如果在二级页表中找到了对应的页表项，并且存在位为 1，表示该虚拟页面在物理内存中有对应的物理页面。
- 此时，可以根据页表项中的物理页面地址和虚拟地址的低位部分（页内偏移量）计算出物理地址。
- 物理地址 = 物理页面地址 + 页内偏移量。

页面置换：

- 如果在页表查找过程中发现虚拟页面不在物理内存中（存在位为 0），则发生页面缺页中断。
- 操作系统需要选择一个物理页面进行置换，将需要的虚拟页面从磁盘加载到物理内存中，并更新页表。
- 页面置换算法可以根据不同的需求选择，如先进先出（FIFO）算法、最近最少使用（LRU）算法、Clock页替换算法等。

页表更新：

- 当进程进行内存访问时，如果对虚拟页面进行了写操作，操作系统需要将该页面标记为脏页，并在适当的时候将脏页写回磁盘。
- 当进程退出或发生上下文切换时，操作系统需要更新页表，以确保页表的正确性。

页面替换算法

- **FIFO（先进先出）：**淘汰最先进入内存的页 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，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。 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。
- **时钟（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。
- **LRU（最近未被使用）：**LRU算法的核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。基于这个假设，算法会优先淘汰那些最近最少被访问的数据，以便为新的数据腾出空间。其主要应用场景是缓存。在缓存系统中，通过保留最近使用的数据而淘汰最久未使用的数据，可以提高缓存的命中率，从而提升系统性能。LRU算法的实现通常基于一个双向链表和一个哈希表。双向链表用于维护数据的访问顺序，哈希表则用于快速查找数据。当需要获取某个数据时，首先在哈希表中查找。如果数据存在，将其从双向链表中移动到链表头部（表示最近使用），并返回数据的值。如果数据不存在，则返回缓存未命中的标志。当需要插入新数据时，首先在哈希表中查找。如果数据已经存在，则更新数据的值，并将其从双向链表中移动到链表头部。如果数据不存在，则插入新数据到双向链表的头部，并在哈希表中添加对应的映射。如果插入后缓存容量超过限制，则从双向链表尾部移除最久未使用的数据，并在哈希表中删除对应的映射。
  - **与FIFO对比：**
    1. **更准确地反映页面使用情况**：
       - FIFO算法按照页面进入内存的顺序进行置换，不考虑页面的实际使用情况。因此，它可能会淘汰掉一些频繁使用的页面，导致缓存命中率下降。
       - LRU算法则根据页面最近的使用情况来决定哪些页面应该被淘汰。它认为最近未被使用的页面在未来被使用的可能性较低，因此优先淘汰这些页面。这种策略能够更准确地反映页面的使用情况，从而提高缓存命中率。
    2. **减少抖动现象**：
       - FIFO算法在处理某些程序时，可能会出现抖动现象，即频繁地在内存和磁盘之间交换页面，导致系统性能下降。
       - LRU算法由于能够更准确地反映页面的使用情况，因此能够减少抖动现象的发生，提高系统的稳定性和性能。
  - **与时钟算法相比：**

    1.更高的缓存命中率：
    - 时钟算法通过循环扫描页面并检查访问位来决定哪些页面应该被淘汰。虽然它能够在一定程度上反映页面的使用情况，但相比LRU算法仍然存在一定的不足。

- LRU算法通过维护一个双向链表来记录页面的使用顺序，能够更精确地反映哪些页面是最近被使用的，从而具有更高的缓存命中率。

2.更好的适应性：

- 时钟算法在扫描页面时，如果访问位为1，则会将其置为0并继续扫描下一个页面。这种策略在处理某些程序时可能会不够灵活。
- LRU算法则能够根据页面的实际使用情况动态地调整页面的顺序，从而更好地适应不同的程序和工作负载。