OS_lab3实验报告

组员：刘俊彤，刘玉菡，孙启森

练习 1：理解基于 FIFO 的页面置换算法

在 FIFO 页面置换算法下，从一个页面被换入到被换出的过程涉及到以下函数和宏的处理：

1. `_fifo_init_mm`：
   - 初始化页面置换算法所需的数据结构（如队列）并将 `mm_struct` 的 `sm_priv` 指向这个队列。这确保了页表项可以被正确追踪。
2. `_fifo_map_swappable`：
   - 当页面首次被载入到内存中时，该函数被调用来将新页面添加到队列的尾部，符合 FIFO 算法的操作原理，即最近载入的页面放在队列尾部。
3. `list_add` (宏)：
   - 在 `_fifo_map_swappable` 中调用，用于将新页面链接到队列中。这实际上是将页面加入到内存管理的数据结构中。
4. `_fifo_swap_out_victim`：
   - 当需要在内存中为新页面腾出空间时，此函数负责选择并移除队列头部的页面（即最早进入内存的页面），实现页面的置换。
5. `list_prev` (宏)：
   - 在选择要置换出去的页面时使用，用于获取队列头部的页面。
6. `list_del` (宏)：
   - 用于从队列中移除被选为牺牲页面的页面项，即断开其在链表中的链接。
7. `le2page` (宏)：
   - 在 `_fifo_swap_out_victim` 中用于从链表元素转换回对应的页面结构体，这是进行页面管理的关键转换。
8. `_fifo_check_swap`：
   - 测试函数，用于验证 FIFO 页面置换算法是否正常工作。通过模拟页面访问和置换来检查算法是否如预期那样操作。
9. `list_init`
   - 初始化 `pra_list_head`，这是一个双向链表头，用于跟踪内存中所有可换出页面的顺序。
10. `_fifo_tick_event`：
    - 用于模拟时钟中断，在FIFO算法中可能不进行操作，但在一个完整的系统中，它可以用于周期性地检查或调整内存使用情况。
11. `_fifo_set_unswappable`：
    - 用于设置某些页面为不可交换，虽然在FIFO算法中通常所有页面都是可交换的。这个函数可以为未来的拓展保留，或用于特定条件下阻止页面被换出。
12. `assert` (宏)：
    - 用于验证操作过程中的状态，确保页面置换和访问行为符合预期，如在 `_fifo_check_swap` 中检查置换次数是否正确。

这些函数和宏共同实现了 FIFO 算法的核心机制：保持一个页面队列，新页面加入队尾，内存不足时从队首移除最早的页面。

练习 2：深入理解不同分页模式的工作原理

get_pte() 函数解析:

- `get_pte()` 函数在页表中查找或创建页表项，关键于实现虚拟地址到物理地址的映射。这个函数的工作原理在于处理虚拟内存地址，确保每个地址都能映射到一个物理页上，或在映射不存在时创建这样一个映射。

两段形式类似的代码解释:

- `get_pte()` 函数中包含形式类似的代码段主要因为处理不同层级的页表（如 sv32, sv39, sv48）时，操作逻辑相似。每个级别的页表都可能需要从物理内存中分配一个新的页表页面，然后初始化这个新页，并更新上级页表以包含对新页表的引用。
    - **相似之处**：每个级别都涉及查找或创建页表项。如果当前级别的页表项不存在（即未分配内存），则需要分配一个新的页表页面，并在该级别的页表中创建新的页表项。
    - **不同之处**：主要在于操作的页表级别不同，地址解析的深度不同，这影响了访问具体页表项的索引计算方式。

是否应该将查找和分配功能拆开:

优点:

- **简化调用**：将查找和分配合并简化了外部调用者的逻辑，使得外部只需一个函数调用即可处理页表项的获取问题，不需要关心内部的查找与分配逻辑。
- **效率**：在进行页表项查找时直接进行分配操作可以减少一次函数调用的开销，因为通常查找和分配是连续的操作。

缺点:

- **函数复杂度高**：函数内部逻辑复杂，职责过多，既要处理查找也要处理分配，这增加了维护的难度和出错的风险。
- **灵活性降低**：硬编码了查找和分配的逻辑，当分页策略需要调整时，修改更加困难。

是否有必要拆开:

- **提高可读性与可维护性**：拆分函数可以使每个函数的职责更加单一，便于理解和维护。
- **增加灵活性**：单独的查找和分配函数可以根据不同的需要灵活调用，更好地适应不同场景的需求，如只查找不分配。

练习 3：给未被映射的地址映射上物理页

设计实现过程

```
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;
    }
    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;
    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;
            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;
}
```

1. **寻找 VMA**：
   - 使用 `find_vma` 函数寻找包含出错地址 `addr` 的虚拟内存区域（VMA）。如果找不到，或地址不在任何 VMA 范围内，则报错。
2. **权限设置**：
   - 根据 VMA 的权限设置页表项的权限。如果 VMA 允许写入，则在权限中添加写权限。
3. **地址对齐**：
   - 将触发缺页异常的地址向下对齐到页边界。
4. **获取或创建页表项**：
   - 使用 `get_pte` 函数获取地址对应的页表项（PTE）。如果页表不存在，该函数还会创建新的页表。
5. **页映射**：
   - 如果页表项为空（即页面未映射），调用 `pgdir_alloc_page` 函数分配一个新页面，并将其映射到出错地址。

- 如果页表项已存在且标记为交换出的页，通过 `swap_in` 函数从磁盘加载页面内容到物理内存，并重新建立映射。

6. **更新交换管理**：
   - 如果启用了交换机制（`swap_init_ok`），则更新页面的可交换信息，使得该页面以后可以被交换出内存。

页目录项（PDE）和页表项（PTE）的作用：

- PDE 和 PTE 是构成多级分页机制的核心。PDE 指向页表，而 PTE 指向实际的物理内存页。在页替换算法中，PTE 中的状态位（如存在位、脏位）可以用来确定页面是否被修改过，是否需要写回磁盘，或者是否可以直接从内存中替换出去。

缺页服务例程中的硬件操作：

- 当出现缺页异常时，硬件自动将引起异常的虚拟地址加载到 CR2 寄存器，异常错误码被推送到栈上。硬件还会停止当前指令的执行，转而执行操作系统中的异常处理程序（如 `do_pgfault`）。

Page 结构与页表项的对应关系：

- `Page` 结构体的数组每一项都对应于一个物理页。页表中的 PTE 通过物理页号（PPN）与 `Page` 结构体数组中的项相对应，即每个 PTE 指向 `Page` 数组的一个特定项，这样可以通过 PTE 管理和访问实际的物理内存页。

练习 4：补充完成 Clock 页替换算法

设计实现过程

1. **初始化（`_clock_init_mm`）**：
   - 初始化用于 Clock 算法的页面列表 `pra_list_headc` 和当前指针 `curr_ptr`。
   - `curr_ptr` 指向链表的头部，作为时钟指针的起始位置。
2. **映射可换出页面（`_clock_map_swappable`）**：

```
list_add_before((list_entry_t*) mm->sm_priv,entry);
page->visited = 1;
```

   - 新页面加入链表尾部，并将其访问标志 `visited` 设置为 1（表示已被访问）。
3. **选择换出页面（`_clock_swap_out_victim`）**：

```
curr_ptr = list_next(curr_ptr);
        if(curr_ptr == head)
        {
            curr_ptr = list_next(head);
            if(curr_ptr == head)
            {
                *ptr_page = NULL;
                return 0;
            }

        }
        struct Page* page = le2page(curr_ptr, pra_page_link);
```

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

- 使用时钟指针（`curr_ptr`）遍历链表。
- 如果页面的 `visited` 标志为 1，则清除该标志并移动指针到下一个页面。
- 如果页面的 `visited` 标志为 0，则选择该页面作为换出的牺牲页面，从链表中删除，并更新 `curr_ptr`。

4. **检查和验证（`_clock_check_swap`）**：
   - 这个函数用于测试算法的实际效果，确保页面置换按预期工作。

Clock 算法与 FIFO 算法的比较

- **FIFO 算法**：
  - 简单、直观，总是选择最先进入内存的页面进行替换。
  - 缺点是可能将经常访问的页面换出，特别是在页面刚好是最早载入而经常被访问的情况下。
  - 存在 Belady 异常，即可用页面帧数增加时，缺页率可能上升。
- **Clock 算法**：
  - 是 FIFO 的改进，通过为每个页面引入访问位（visited），给页面第二次机会。
  - 当页面第一次被访问时，不立即将其换出，而是清除访问位并继续寻找下一个候选页面。
  - 只有在页面的访问位被清零后，该页面仍被选中时，才进行替换。
  - 减少了不必要的页面置换，特别是对于那些被频繁访问但早期载入的页面。
  - 不会出现 Belady 异常，因为它更智能地处理了页面的访问历史。

练习 5：页表映射方式相关知识

好处与优势

使用大页页表映射方式可以减少页表项的数量，从而降低翻译后备缓冲器（TLB）的压力，并提高TLB的命中率，因为每个TLB条目可以覆盖更大的内存区域。这种方法还减少了页表遍历的层数，提高了内存访问效率，并可能通过增强数据的局部性来优化缓存利用率。

坏处与风险

大页页表映射可能导致内存浪费，尤其是在应用程序不能完全使用分配的大页内存时，会产生内存碎片。此外，管理不同大小页面增加了内存管理的复杂性，需要操作系统和硬件的额外支持。大页的灵活性较低，不适合需要频繁动态分配和释放小块内存的环境。