lab 3 缺页异常和页面置换

小组成员：徐亚民，肖胜杰，张天歌

练习0：填写已有实验

本实验依赖实验1/2。请把你做的实验1/2的代码填入本实验中代码中有“LAB1”,“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）

> 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？
>
> - 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。

1.1 流程梳理

首先梳理一下整体的工作流程，从初始化到页面置换，再到异常处理，逐步调用相关函数和宏以实现页面替换：

1. 初始化阶段

- 操作系统启动时，`kern_init()`函数负责初始化内存管理、虚拟内存管理等。这里调用了`swap_init()`，来初始化页面置换算法。
- `swap_init()`调用了`swapfs_init()`检查页面交换文件系统，随后将`swap_manager`指向具体的页面置换算法实现（例如 FIFO）。若初始化成功，进入`check_swap()`验证交换功能。

2. 异常处理阶段

- 系统在运行时如果遇到页面访问异常，会通过`exception_handler()`进入`pgfault_handler()`处理页面错误。
- 在`pgfault_handler()`中，先调用`print_pgfault()`输出异常原因，然后进入`do_pgfault()`进行页面置换处理。

3. 页面置换的核心过程（`do_pgfault()`）

- **换入页面**：

    - 在 `do_pgfault()` 函数中，首先为待换入的页面做准备，获取页面对应的虚拟地址（通过 `find_vma()`）和页表项指针（通过 `get_pte()`）。
    - 调用 `swap_in()` 开始换入操作，先用 `alloc_page()` 分配一个新页面，然后判断是否需要换出其他页面以腾出空间。
    - 如果要将页面从磁盘换入内存，会通过 `swapfs_read()` 将数据写入内存。
    - 接着，通过 `page_insert()` 将新的页面插入页表，这里调用了 `page_ref_inc()`、`page_ref_dec()` 等函数来调整页面引用计数，并利用 `tlb_invalidate()` 刷新 TLB（即清除缓存的虚拟-物理地址映射），以确保新页面生效。
    - 最后，`swap_map_swappable()` 使页面可交换。通过链表结构维护页面队列，配合 `swap_out_victim()` 实现先进先出（FIFO）页面置换策略。
- **换出页面**：
    - 如果在 `alloc_page()` 时判定内存不足，需要腾出空间，则调用 `swap_out()` 进行换出操作。
    - 在 `swap_out()` 中，通过 `swap_out_victim()` 找到待换出的页面，并获取其虚拟地址对应的页表项。
    - 调用 `swapfs_write()` 将页面内容写到磁盘。若成功，则调用 `free_page()` 释放页面；若失败，调用 `swap_map_swappable()` 恢复页面状态。

1.2 函数功能描述

接下来，对关键的 11 个函数进行功能描述：

1.1 `swap_init()`

`swap_init()` 函数的作用是初始化页面替换算法。它首先调用 `swapfs_init()` 函数，以设置模拟硬盘的最大页偏移数并进行检查。接着，设置 `swap_manager` 指针，指定 FIFO 为页面替换算法。初始化成功后，会调用 `check_swap()` 检查页面交换功能。

1.2 `exception_handler()` 与 `pgfault_handler()`

`exception_handler()` 是异常处理的入口点，负责处理页面异常（如读取和存储页面缺页）。当缺页异常发生时，它会调用 `pgfault_handler()`，而 `pgfault_handler()` 再调用 `do_pgfault()` 处理缺页，将地址与内存建立映射。

1.3 `do_pgfault()`

`do_pgfault()` 是页面替换的核心，它首先通过 `find_vma()` 查找虚拟内存管理结构体中的虚拟地址区域，再通过 `get_pte()` 获取或创建页表项。接着调用 `swap_in()` 将需要的页换入内存，并通过 `page_insert()` 完成物理页和虚拟页的映射，最后标记该页为可交换。

1.4 `swap_in()`

`swap_in()` 将页面从硬盘加载到内存。它首先为缺页的虚拟地址分配物理页，接着使用 `get_pte()` 获取页表项，再调用 `swapfs_read()` 将数据从“硬盘”加载到内存中。

1.5 `swapfs_read()` 与 `swapfs_write()`

这两个函数负责对硬盘的读写，分别在页面换入和换出时调用。`ide_read_secs()` 和 `ide_write_secs()` 使用 `memcpy()` 模拟硬盘到内存的数据拷贝。

1.6 `alloc_pages()`

`alloc_pages()` 函数负责分配空闲的物理页，若页面已满，则通过 `swap_out()` 换出页。这个函数也是 `kmalloc()` 的底层支撑。

1.7 `page_insert()`

`page_insert()` 建立物理页和页表项的映射，同时管理页面引用计数 `ref`。映射后，它调用 `tlb_invalidate()` 刷新 TLB，以确保新映射的生效。

1.8 `get_pte()`

`get_pte()` 函数用于构建虚拟地址与页表项之间的映射，确保虚拟地址对应的页表项存在（若不存在，则创建一个新的）。其工作流程为：

1. 获取一级页目录项指针 `pdep1`，并检查其有效性（`PTE_V` 位）。
2. 如果无效且需要创建，则分配新页并更新目录项。
3. 获取二级页目录项指针 `pdep0`，同样检查其有效性。
4. 如果无效且需要创建，则分配新页并更新。
5. 最后，返回虚拟地址对应的页表项指针。

1.9 `swap_map_swappable()`

该函数负责将页设置为可交换状态。在 `FIFO` 算法中，通过 `Page` 结构体中的 `pra_page_link` 成员来管理一个链表，以此实现先进先出的页替换机制：

1. 使用 `list_add` 函数将新的页面加入链表头部。
2. 后续换出操作可以基于该链表进行 FIFO 策略的页面替换。

1.10 `swap_out()`

`swap_out()` 是内存换出到硬盘的顶层接口，其执行流程为：

1. 使用 `swap_out_victim` 选择需要换出的页。
2. 检查页表项权限是否有效。
3. 使用 `swapfs_write` 进行页数据的写入。如果写入成功，则释放该页并更新页表项。
4. 调用 `tlb_invalidate` 以使 TLB 缓存失效。
5. 该函数可以处理多页换出，但当前实现仅能处理单页换出。

1.11 `swap_out_victim()`

`swap_out_victim()` 函数负责选择要换出的页面。对于 `FIFO` 策略，其实现逻辑为：

1. 通过链表尾部节点找到需要换出的页面。
2. 将该节点的页面存入 `ptr_page` 指向的地址。
3. 删除该链表节点，完成页的选择。

练习2：深入理解不同分页模式的工作原理（思考题）

练习3：给未被映射的地址映射上物理页（需要编程）

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`）。
> 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。

4.1 Clock页替换算法设计实现

```
static int
_clock_init_mm(struct mm_struct *mm)
{
     // 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;
}
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);
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    assert(entry != NULL && curr_ptr != NULL);
    // 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(head->prev, entry);
    page->visited = 1;
    return 0;
}

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 2211123*/
        // 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
        curr_ptr = list_next(curr_ptr);
        if(curr_ptr == head)
            curr_ptr = list_next(curr_ptr);

        struct Page *page = le2page(curr_ptr, pra_page_link);
        if(!page->visited) {
            cprintf("curr_ptr 0xffffffff%x\n", curr_ptr);
            list_del(curr_ptr);
            *ptr_page = page;//将该页面指针赋值给ptr_page作为换出页面
            break;
        } else {
            page->visited = 0;
        }
    }
    return 0;
```

```
53 }
```

初始化（`_clock_init_mm`）与 **FIFO** 逻辑相同，**页面置换准备**时需要注意：当一个新的页面需要被标记为“可换出”时，`_clock_map_swappable` 将页面插入到链表的末尾（`pra_list_head` 的前面），模拟 FIFO 的队列结构。同时需要设置页面的 `visited` 标志位为 1，表示页面刚刚被访问过，以此保证其不会立刻被换出。

页面换出（`_clock_swap_out_victim`）：当内存中需要释放一个页面时，`_clock_swap_out_victim` 从 `curr_ptr` 开始扫描链表，找到合适的页面换出。**在遍历到链表末尾后，会回到链表头部继续扫描，直到找到一个合适的页面换出为止。**遍历过程中：

- 如果页面的 `visited` 标志为 0，说明页面未被访问，适合换出，将其从链表中删除并返回该页面。
- 如果页面的 `visited` 标志为 1，说明页面近期被访问过，将其 `visited` 标志置为 0，并继续扫描下一个页面。

```
make[1]: 进入目录“/home/tiange/桌面/OS/riscv64-ucore-labcodes/OperatingSystem/la
b3” + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc ker
n/debug/panic.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/dr
iver/ide.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/
intr.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/vmm.c + cc
kern/mm/swap_fifo.c + cc kern/mm/swap.c + cc kern/mm/default_pmm.c + cc kern/mm/
swap_clock.c + cc kern/mm/pmm.c + cc kern/fs/swapfs.c + cc libs/string.c + cc li
bs/printfmt.c + cc libs/readline.c + cc libs/rand.c + ld bin/kernel riscv64-unkn
own-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img make[1]: 离开目录
“/home/tiange/桌面/OS/riscv64-ucore-labcodes/OperatingSystem/lab3”
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=20494
<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check pmm:                                 OK
  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
Total Score: 45/45
tiange@tiange-virtual-machine:~/桌面/OS/riscv64-ucore-labcodes/OperatingSystem/l
ab3$
```

4.2 CLOCK 与 FIFO 算法的比较

1. **基本策略**:
    - **FIFO 算法**：选择最早进入内存的页面进行换出，即不管页面是否被访问过，直接淘汰。
    - **CLOCK 算法**：在最早页面被换出之前，会检查其 `visited` 标志位。如果该标志为 1，则重置为 0并跳过，直到找到一个 `visited` 为 0 的页面进行换出。
2. **优缺点对比**:
    - **FIFO**：实现简单，但存在**Belady 异常**问题，即增加页面帧数量反而导致缺页率上升，因为它不考虑页面的访问情况。
    - **CLOCK**：相对简单且改进了 FIFO，通过判断 `visited` 标志来决定是否立即换出，使得被频繁访问的页面有更长的留存时间。因此，CLOCK 算法是一种**近似的 LRU (Least Recently Used) 算法**，性能优于 FIFO，但实现较为复杂。

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？

扩展练习 Challenge：实现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。