Lab3实验报告

小组成员： 钟坤原 邢清画 田晋宇

练习1：理解基于FIFO的页面替换算法（思考题）

在FIFO页替换算法的实现中，主要涉及以下关键函数及其作用：

1. `swap_init()`（`kern/mm/swap.c`）：初始化交换系统。调用 `swapfs_init` 来初始化交换文件系统，并检查最大交换偏移量范围，设置交换管理器等。
2. `swap_map_swappable`（`kern/mm/swap.c`）：将页面标记为可交换，将 `mm`、`addr`、`page` 和 `swap_in` 参数传递给交换管理器的 `map_swappable` 函数。
3. `_fifo_map_swappable`（`kern/mm/swap_fifo.c`）：在FIFO页面置换算法中，将页面标记为可交换，并添加到页面队列末尾。
4. `swap_out`（`kern/mm/swap.c`）：将页面从内存交换至磁盘。在FIFO算法中，此函数用于将被置换出的页面写入磁盘的交换分区，以腾出内存空间，同时更新TLB。
5. `swap_tick_event`（`kern/mm/swap.c`）：处理时钟中断事件，将 `mm` 参数传递给交换管理器的 `tick_event` 函数，用于定期更新页面交换系统。
6. `swap_set_unswappable`（`kern/mm/swap.c`）：将页面设置为不可交换，将 `mm` 和 `addr` 参数传递给 `set_unswappable` 函数，以限制页面被换出。
7. `page_insert`（`kern/mm/pmm.c`）：映射物理页面到虚拟地址上，并设置权限。通过 `get_pte` 获取页表项指针 `ptep`，并更新映射关系，调用 `tlb_invalidate` 刷新 TLB 缓存确保立即生效。
8. `alloc_pages`（`kern/mm/pmm.c`）：用于分配物理页面。在FIFO算法中，当没有空闲页时调用 `swap_out` 函数将部分页面换出，以腾出空间来分配新页面。
9. `_fifo_swap_out_victim`（`kern/mm/swap_fifo.c`）：从FIFO队列中选择最早到达的页面作为牺牲页面，将其交换到磁盘并从队列中删除。如果队列为空，则将 `ptr_page` 设为 `NULL`。
10. `tlb_invalidate`（`kern/mm/pmm.c`）：当页表映射关系改变时，刷新 TLB 缓存，以确保新的映射关系生效。由于硬件实现了 TLB 缓存，调用 `flush_tlb` 函数以完成刷新操作。
11. `swap_in`（`kern/mm/swap.c`）：将页面从磁盘换入内存。在FIFO算法中，此函数用于将置换出的页面加载到内存中，通过找到一个物理页面并读取硬盘数据。
12. `swap_init_mm`（`kern/mm/swap.c`）：初始化交换系统的内存管理器，通过 `init_mm` 函数将 `mm` 参数传递给交换管理器。

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？*

2.1 get_pte()代码的相似性原因

首先分析sv32，sv39，sv48之间的异同:

1. **虚拟地址空间大小：**
   - Sv32：32 位虚拟地址空间，使用两个虚拟页号（VPN）和一个偏移量。
   - Sv39：39 位虚拟地址空间，分为 4KiB 页，用于支持 512GB 虚拟地址空间的系统，采用三级页表。
   - Sv48：48 位虚拟地址空间，比 Sv39 多一个页表级别，支持 256TB 虚拟地址空间，但增加了页表存储开销和延迟。
2. **页表级别：**
   - Sv32：两级页表。
   - Sv39：三级页表，支持 4KB 页、2MB 大页和 1GB 超大页。
   - Sv48：四级页表，兼容 Sv39，增加一个页表级别以支持更大的虚拟地址空间。
3. **地址转换和内存保护：**
   - Sv32：两级页表实现地址转换。
   - Sv39：三级页表实现地址转换，详细定义在 RISC-V 手册中。
   - Sv48：类似 Sv39，但多一个页表级别，增加了转换的复杂度。

这两段代码的相似性源于多级页表结构的本质，以及在虚拟地址到物理地址的转换过程中逐级查找页表的通用逻辑。以下是这两段代码相似性的原因的分析：

1. **多级页表机制的结构性特点：**
   - 在多级页表（如 Sv39 的三级页表）中，虚拟地址被分成多个部分，每一部分用于查找相应的页表级别。具体来说，Sv39 的虚拟地址分成了 VPN[2]、VPN[1] 和 VPN[0]，每个部分分别对应不同的页表级别（从顶层到底层）。
   - 代码的逻辑体现了这种多级结构，每次迭代会进入一个新的页表级别，最终找到实际的页表项。
2. **逐级查找和递归分配的必要性：**
   - 在多级页表中，每一级页表项指向下一级页表的物理地址。要找到最终的物理地址，需要依次从顶层页表逐级查找直到底层页表。
   - 代码通过 `pdep1` 和 `pdep0` 等变量逐级获取不同页表级别的页表项，并检查其有效性。如果页表项无效（即不存在有效的物理地址映射），则会为该页表项分配新的页并初始化。这种逐级检查和分配确保了页表结构的完整性。
3. **虚拟地址的分段和页表索引机制：**
   - 在 Sv39 的三级页表中，虚拟地址的高位用于索引顶层页表项，低位依次用于索引下一级的页表。这种设计使得代码结构具有一致性：通过宏 `PDX1(la)`、`PDX0(la)` 等，代码可以自动提取对应的索引并查找每一级页表的项。
   - 每一层的查找逻辑几乎相同，仅是针对不同的索引，因此代码段中的逻辑是高度相似的。
4. **内存保护和页表项有效性检查：**
   - 在每一级页表查找时，代码都需要检查当前页表项是否有效（即是否具有有效的物理地址映射）。如果无效，则会分配新的页并设置相关标志（如 PTE_V 表示有效、PTE_U 表示用户态等）。

- 这种有效性检查和页分配逻辑在每一级页表项查找时都是相同的，保证了无论在哪一级页表中，系统都能处理缺页并建立新的映射关系。

5. **代码的递归模式和扩展性：**
   - 代码的逻辑结构可以轻松扩展到其他页表模式，如 Sv32 和 Sv48。对于 Sv32，由于只有两级页表，因此只需检查两级页表项并返回结果。而对于 Sv48，则需要多增加一层 `pdep2` 的逻辑，按照同样的方式逐层查找和分配。
   - 这种递归模式使得代码在应对不同级别的页表（Sv32、Sv39 和 Sv48）时可以保持一致的逻辑结构，因此代码在多级页表结构中表现出相似性。

2.2 get_pte()将页表项将查找和分配合并的原因

目前 `get_pte()` 函数将页表项的查找和分配合并在一个函数里，我们小组认为这种写法是合理的，且没有必要将两个功能拆开，具体原因如下：

优点

1. **简化接口，提高代码使用便捷性**
   - 将查找和分配合并在一个函数中，可以让外部调用方无需在多个函数间切换。例如在发生缺页异常时，`get_pte()` 可以直接返回所需的页表项指针，减少外部调用方的操作步骤，保持接口的简洁。这种设计让代码更加直观、易于使用，降低了出错的可能性。
2. **确保操作的原子性，避免竞态条件**
   - 合并查找和分配操作可以确保整个过程是原子的。尤其是在多线程或并发环境下，这种设计有助于避免因查找后未及时分配而导致的竞态条件问题。这样，`get_pte()` 可以在一次调用中保证查找和分配的完整性，提升代码的稳定性和可靠性。
3. **提升性能，减少不必要的内存访问**
   - 合并后的 `get_pte()` 可以在逐级查找页表项的过程中避免重复访问同一个页表项。在多级页表系统中，每级页表的访问都是一次内存操作，将查找和分配合并可以有效减少内存访问次数，从而提高性能。

缺点

1. **难以定位缺失的页表级别**
   - 如果将查找和分配操作分开，查找函数返回 `NULL` 时只能表明存在缺失的页表项，但无法精确定位是在哪一层出现缺失。这样，外部调用方在处理缺失情况时需要逐级检查并手动分配页表项，从而增加了代码的复杂性和额外的查找开销。
2. **可维护性和复用性降低**
   - 将查找和分配功能合并在一起，使得 `get_pte()` 函数具备了双重功能，违背了单一职责原则。尽管如此，在实际使用中，查找和分配往往是连续的操作，将它们合并可以减少调用的复杂性。然而，如果未来某些场景仅需要查找而不需要分配，这种合并会降低代码的复用性，可能需要额外的重构。

在缺页异常处理函数 `do_pgfault()` 中，通过 `get_pte()` 可以直接获取缺页的虚拟地址对应的页表项，并检查该页表项是否为空（即 `*ptep == 0`），以确认是否需要分配新的物理页。如果将查找和分配功能拆分，那么 `do_pgfault()` 在调用查找函数时，如果遇到页表项缺失，调用方需要逐级检查哪个层级缺失并手动分配，这将显著增加代码的复杂度。

此外，由于多级页表的查找和分配是递归、逐级的操作，每一级的检查和分配都需要维护一致性，合并在一个函数内可以直接在每一级检测缺失并进行分配，从而减少不必要的重复查找开销，提高效率。

结论

综上所述，`get_pte()` 函数将查找和分配合并在一起，可以简化接口设计、提升性能，并确保多级页表操作的原子性。特别是在多级页表结构中，这种合并的设计便于在查找过程中逐级检测并分配缺失的页表项，避免了外部调用方的额外负担。我们认为这种写法合理，没有必要将查找和分配拆分为两个函数，拆分反而可能增加不必要的复杂性和性能开销。

练习3：给未被映射的地址映射上物理页（需要编程）

```
//(1) According to the mm AND addr, try
//to load the content of right disk page
swap_in(mm,addr,&page);
//into the memory which page managed.
//(2) According to the mm,
//addr AND page, setup the
//map of phy addr <--->
page_insert(mm->pgdir,page,addr,perm);
//logical addr
//(3) make the page swappable.
swap_map_swappable(mm,addr,page,1);
```

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

- 该行调用 `swap_in` 函数，将地址 `addr` 对应的磁盘页加载到内存中，并存储在 `page` 中。
- `swap_in` 通过 `mm` 和 `addr` 定位磁盘上存储的页面，将其内容读取到一个新的物理页面中（即 `page` 指向的内存区域）。
- `&page` 是一个指向 `page` 结构的指针，用于接收从磁盘加载的页面在内存中的地址。

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

- 这行代码使用 `page_insert` 函数在页表中建立物理页面和逻辑地址的映射。
- `mm->pgdir` 是当前进程的页目录，`page` 是加载的物理页面，`addr` 是逻辑地址，`perm` 是权限。
- `page_insert` 函数会在 `mm->pgdir` 中为 `addr` 设置指向 `page` 的映射，使得进程可以通过逻辑地址 `addr` 访问刚加载的页面内容，同时设置页面的权限（读写等）。

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

- 这行代码使用 `swap_map_swappable` 函数将页面标记为可交换（swappable），使得该页面可以在需要时被换出到磁盘。
- 参数 `1` 通常用于指定该页面是否立即参与换出策略。
- `swap_map_swappable` 函数会将 `addr` 和 `page` 添加到进程的可交换页面管理结构中，以便在内存不足时可以将该页面交换到磁盘，进行内存管理优化。

问题：

1. 页目录项（Page Directory Entry）和页表项（Page Table Entry）对 ucore 实现页替换算法的潜在用处

1.用来标记该页是否在内存中。如果一个页不在内存中且访问发生时触发缺页异常，系统可以通过页替换算法在内存中为该页腾出空间。

2.页表项中的权限位控制对该页的访问权限，例如只读或读写。这对页替换算法非常重要，因为某些页面（如代码页）可以标记为只读，以减少被替换的可能性。

3.帮助页替换算法判断页面的最近访问和修改情况。访问位用于跟踪页面是否被访问过，而修改位用于指示页面是否在内存中被修改。页替换算法（如 LRU、Clock、LRU-Approximation 等）可以使用这些位来选择适合换出的页面。

2. 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，硬件要做哪些事情？

当 ucore 的缺页服务例程在执行过程中出现页访问异常时，硬件会进行以下操作：

1. **保存上下文**：硬件将保存当前的处理器状态，包括程序计数器（PC）、CPU 寄存器等，以便在处理完异常后能够恢复执行。
2. **设置异常原因码**：硬件将标记发生页访问异常的原因，比如无效页访问、权限不足等，并将该异常原因代码写入控制寄存器中。
3. **存储访问地址**：硬件会将引起异常的地址存储到指定的寄存器（例如 CR2 寄存器），以供操作系统的异常处理例程使用。
4. **中断当前指令**：硬件将中断正在执行的指令，并将控制权交给操作系统，使其可以处理异常。这通常会引发上下文切换，使得操作系统能够响应异常。

3. 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，数据结构 `Page` 的全局变量数组与页表中的页目录项和页表项之间存在对应关系：

页表项和页目录项存储的结构体：

```
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)
};
```

其中使用了一个`visited`变量，用来记录页面是否被访问。
在`map_swappable`函数，我们把换入的页面加入到FIFO的交换页队列中，此时页面已经被访问，visited置为1.在`clock_swap_out_victim`，我们根据算法筛选出可用来交换的页面。在CLOCK算法我们使用了visited成员：我们从队尾依次遍历到队头，查看visited变量，如果是0，则该页面可以被用来交换，把它从FIFO页面链表中删除。由于PTE_A 表示内存页是否被访问过，visited与其对应。

练习4：补充完成Clock页替换算法（需要编程）

`_clock_init_mm`

这个函数主要作用是初始化页面链表和当前指针

```
// 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);
```

初始化 `pra_list_head` 为空链表：

```
list_init(&pra_list_head);
```

这行代码调用 `list_init` 函数，将 `pra_list_head` 初始化为一个空的双向链表头。链表用于管理所有被加载到内存中的页面，以便在页面替换算法中追踪页面的顺序。

初始化当前指针 `curr_ptr` 指向 `pra_list_head`：

```
curr_ptr = &pra_list_head;
```

这里将 `curr_ptr` 初始化为指向 `pra_list_head`，表示当前页面替换的位置从链表头开始。`curr_ptr` 在页面替换时会循环遍历链表，指示哪个页面可能会被替换。在时钟页面替换算法中，`curr_ptr` 的位置会随着页面的访问而移动，以找到符合条件的页面进行替换。

将 `mm` 结构的私有成员指针 `sm_priv` 指向 `pra_list_head`：

```
mm->sm_priv = &pra_list_head;
```

`mm->sm_priv` 是 `mm_struct` 中的一个指针，用于保存特定的页面替换算法所需的数据结构。在此处，`sm_priv` 被设置为指向 `pra_list_head`，方便后续算法可以通过 `mm_struct` 访问页面链表。这样做的好处是将页面管理结构与具体的页面替换算法分离，使得替换算法可以通过 `sm_priv` 访问到页面链表。

`_clock_map_swappable`

这个函数主要功能是将新到达的页面按照顺序插入到页面链表的末尾，并标记该页面为已访问。这种方式为页面替换算法提供了页面的时间顺序和访问情况，方便算法在内存不足时根据策略进行页面替换。

获取链表头指针：

```
list_entry_t *head = (list_entry_t*) mm->sm_priv;
```

这行代码将 `mm->sm_priv` 转换为 `list_entry_t` 类型的指针，并赋值给 `head`。由于 `mm->sm_priv` 在初始化时被设置为指向页面链表 `pra_list_head`，所以这里的 `head` 实际上就是页面链表的头指针，方便后续操作访问链表。

将页面插入链表末尾：

```
list_add(head, entry);
```

`list_add` 函数用于将新页面的链表项 `entry` 添加到 `head` 所指向的链表末尾。这样做的目的是将新到达的页面添加到链表尾部，保证页面的访问顺序按时间先后排列，以便后续的替换算法（如 FIFO 或时钟算法）使用这些信息来决定哪个页面需要被替换。

设置页面的访问标志：

```
page->visited = 1;
```

这行代码将页面 `page` 的 `visited` 标志设置为 `1`，表示该页面刚刚被访问过。在时钟页面替换算法中，`visited` 标志用于判断页面的访问情况。当替换算法遍历页面时，如果发现页面的 `visited` 标志为 `1`，则表示页面最近被访问过；否则，页面可能会被优先替换。

`_clock_swap_out_victim`

这个函数实现了时钟页面替换算法的核心部分，通过遍历页面链表查找适合换出的页面：如果找到未被访问的页面（`visited` 为 `0`），则选择它作为换出页面。如果页面已被访问，则重置其 `visited` 标志为 `0`，并继续查找下一个页面。

遍历页面链表，查找最早未被访问的页面：

```
if(curr_ptr == head){
    curr_ptr = list_prev(curr_ptr);
    continue;
}
```

- 这部分代码用于检查 `curr_ptr` 是否指向链表头 `head`，即检查是否已遍历完整个链表一轮。
- 如果 `curr_ptr` 与 `head` 相同，表示已到达链表的起始位置，因此 `curr_ptr` 需要移到链表的前一个位置，继续下一轮的遍历。

获取当前页面的 `Page` 结构指针：

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

- 这行代码将 `curr_ptr` 转换为 `Page` 结构指针 `curr_page`。
- `le2page` 是一个宏，它根据链表项 `curr_ptr` 的地址（`pra_page_link` 字段）获取包含该链表项的 `Page` 结构指针 `curr_page`，以便访问页面的其他属性。

检查并选择未被访问的页面作为换出目标：

```
if(curr_page->visited == 0){
    cprintf("curr_ptr %p\n", curr_ptr);
    curr_ptr = list_prev(curr_ptr);
    list_del(list_next(curr_ptr));
    *ptr_page = curr_page;
    return 0;
}
```

- 这段代码判断 `curr_page` 是否未被访问（即 `visited` 标志为 `0`）。
- 如果 curr_page 的 visited标志为 0，则该页面作为换出目标：
  - 先打印 `curr_ptr` 的地址信息（用于调试）。
  - 将 `curr_ptr` 移到链表的前一个位置。
  - 使用 `list_del` 从链表中删除 `curr_page`。
- 将 `curr_page` 的指针赋值给 `ptr_page`，标记为换出页面，并返回 `0` 表示成功找到换出目标。

重置访问标志并移到下一个页面：

```
curr_ptr = list_prev(curr_ptr);
```

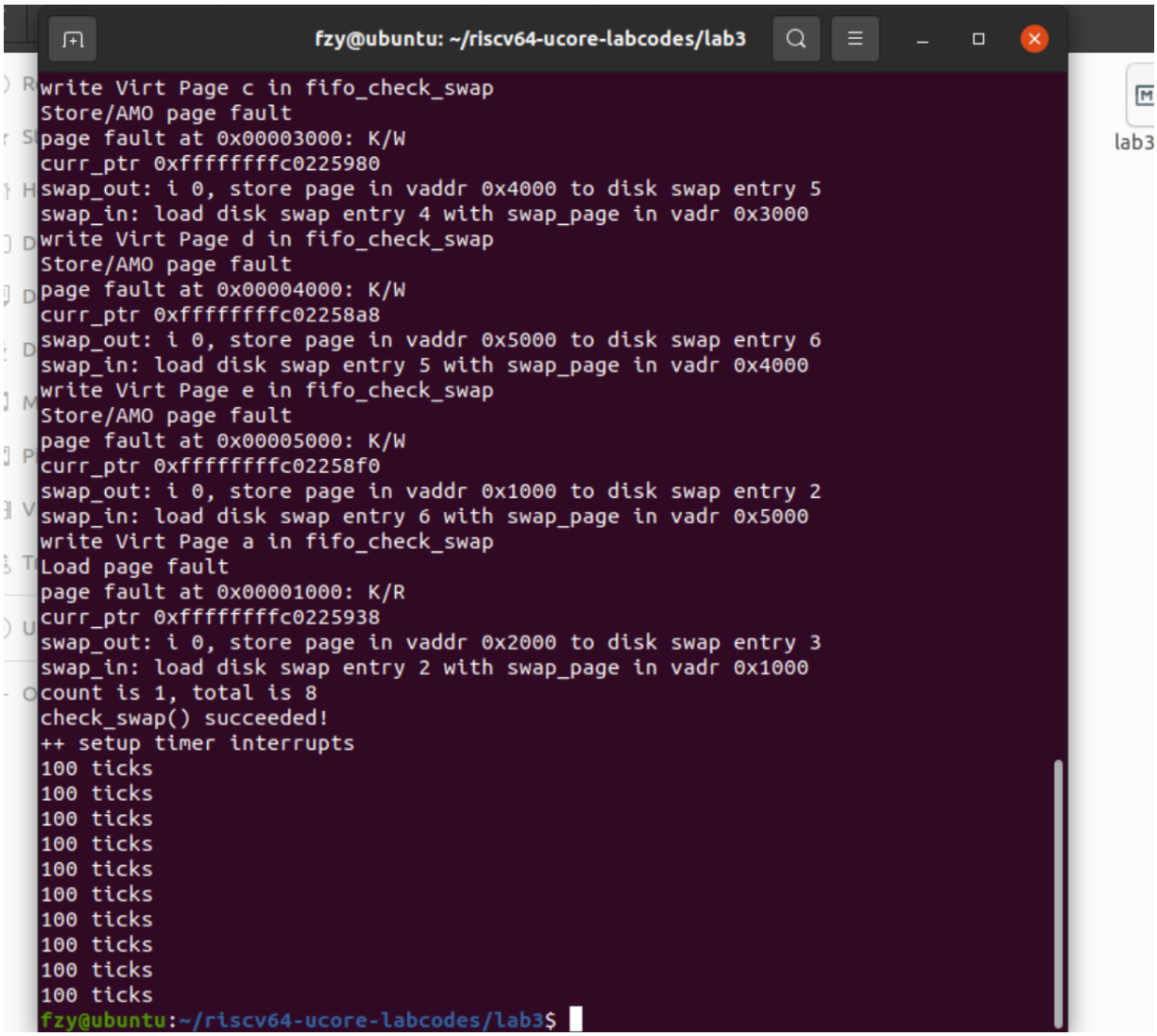
- 如果当前页面已被访问过（即 `visited` 标志为 `1`），则将 `visited` 标志重置为 `0`，表示该页面已经被重新访问。
- 然后，将 `curr_ptr` 移到链表的前一个页面，继续查找未被访问的页面。

不同：

- Clock算法：每次添加新页面时会将页面添加到链表尾部。每次换出页面时都会遍历查找最近没有使用的页面。
- FIFO算法：将链表看成队列，每次添加新页面会将页面添加到链表头部（队列尾部）。每次换出页面时不管队头的页面最近是否访问，均将其换出。

实验结果

make qemu



make grade

```
make[1]: Entering directory '/home/fzy/riscv64-ucore-labcodes/lab3' + cc kern/in
it/entry.S + cc kern/init/init.c + cc kern/libs/stdio.c + cc kern/debug/panic.c
+ cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/driver/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/swap_lru.c + 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 libs/printfmt.c + cc libs/readline.c + cc libs/rand.c + ld bin/kernel risc
v64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img make[1]:
Leaving directory '/home/fzy/riscv64-ucore-labcodes/lab3'
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=25099
<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check pmm:                                 OK
  -check vmm:                                 OK
  -check swap page fault:                     OK
  -check ticks:                               OK
Total Score: 45/45
fzy@ubuntu:~/riscv64-ucore-labcodes/lab3$
```

代码通过测试

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）

采用“一个大页”的页表映射方式相较于分级页表有以下优缺点：

优点

1. **较小的页表**：大页减少了页表的大小，降低了内存占用和管理开销，适合大内存需求的系统。
2. **降低 TLB 缓存失效率**：大页减少了需要缓存的页表条目，从而降低 TLB 失效概率，提升内存访问性能。
3. **更快的地址转换**：大页减少了需要查找的页表级别，硬件可以更快速地完成地址转换。
4. **适合特定工作负载**：在科学计算、多媒体应用等连续内存访问模式的工作负载中，大页能显著提升性能。

缺点

1. **内存碎片**：大页需要连续的物理内存，易造成内存碎片，浪费内存资源。
2. **不适合小型数据结构**：小型数据结构或散乱内存访问模式下，大页可能浪费内存。
3. **不适合多任务系统**：多任务系统中，大页限制了内存的动态分配和共享，不利于满足不同进程的内存需求。
4. **管理复杂性**：大页的管理需更多操作系统支持，增加了页错误处理和页面交换的复杂性。

大页映射方式在特定工作负载下具有性能和管理上的优势，但也带来内存碎片、管理复杂性等问题。分级页表结构在适应性和灵活性方面更具优势，因此更适合多样化的应用场景。

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算法的核心思想是，在内存满时，选择最久未被访问的页面进行交换到磁盘。

1. _lru_init_mm: 初始化操作

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

- 初始化一个链表 `pra_list_head` （初始化pra_list_head为**空链表**）用来记录内存中可交换页面的顺序。
- `pra_list_head` 是一个链表头，表示当前所有可交换的页面。
- 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。

- `mm->sm_priv` 是 `mm_struct` 结构中的一个私有成员，指向链表头 `pra_list_head`，用于后续的页面替换算法操作。

2. _lru_map_swappable: 按照LRU算法将页面加入交换队列

该函数实现了将一个页面添加到LRU链表的尾部，也就是将其标记为“最近访问”。

```
static int _lru_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);

    // 将页面插入到队列的末尾
    list_add_before((list_entry_t*)mm->sm_priv, entry);
    page->visited = 1;

    cprintf("map_swappable: Added page at addr 0x%lx to the swap queue\n", addr);

    return 0;
}
```

- 通过 `list_add_before` 函数，将页面链接 `pra_page_link` 插入到链表 `pra_list_head` 的末尾。
- `page->visited` 设置为 1，表示该页面最近被访问过。
- 调用 `cprintf` 打印调试信息，便于验证页面是否按预期被加入交换队列。

3. _lru_swap_out_victim: 按照LRU算法选择一个待换出的页面

该函数实现了LRU算法中选择一个“最久未使用”的页面进行交换（即从内存移到磁盘）。

```
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);

    while (1) {
        curr_ptr = list_next(curr_ptr);
        if (curr_ptr == head) {
            curr_ptr = list_next(curr_ptr);
            if (curr_ptr == head) {
                *ptr_page = NULL;
                return 0;
            }
        }

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

```
        // 如果页面没有被访问过，说明是最久未访问的页面，选中它并从队列中删除
        if (!page->visited) {
            *ptr_page = page;
            list_del(curr_ptr);

            cprintf("curr_ptr %p\n", curr_ptr);
            cprintf("swap_out: i 0, store page in vaddr 0x%lx to disk swap entry
2\n", (uintptr_t)page);

            return 0;
        } else {
            // 如果页面被访问过，重置访问标志
            page->visited = 0;
        }
    }

    return 0;
}
```

- 首先，遍历链表 `pra_list_head`，`curr_ptr` 指向链表的当前节点。
- 如果页面 `page->visited` 为 0，表示该页面最久未访问，是交换的候选页面。此时，选中该页面，将其从链表中删除，并将其指针返回。
- 如果页面 `page->visited` 为 1，表示该页面最近被访问过，将其访问标志重置为 0，继续检查下一个页面。

4. _lru_check_swap: 检查交换的正确性

```
static int _lru_check_swap(void)
{
    cprintf("write Virt Page c in fifo_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num == 4);
    cprintf("write Virt Page a in fifo_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num == 4);
    cprintf("write Virt Page d in fifo_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num == 4);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num == 4);
    cprintf("write Virt Page e in fifo_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num == 5);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num == 5);
    cprintf("write Virt Page a in fifo_check_swap\n");
    *(unsigned char *)0x1000 = 0x0a;
```

```
    assert(pgfault_num == 6);
    cprintf("write Virt Page b in fifo_check_swap\n");
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num == 7);
    cprintf("write Virt Page c in fifo_check_swap\n");
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num == 8);
    cprintf("write Virt Page d in fifo_check_swap\n");
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num == 9);
    cprintf("write Virt Page e in fifo_check_swap\n");
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num == 10);
    cprintf("write Virt Page a in fifo_check_swap\n");
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num == 11);
    return 0;
}
```

- 模拟一系列的页面访问（读写操作），来检查页面交换机制是否按照预期工作。通过 `pgfault_num` 计数，检查缺页中断的次数，验证是否符合LRU算法的页面替换逻辑。
- 每次访问虚拟页面时，都会打印调试信息并通过 `assert` 验证是否发生了正确的页面交换。

5. LRU Swap Manager 结构体实现

定义了LRU交换管理器。它将LRU算法相关的函数（如初始化、页面交换、检查交换等）注册到 `swap_manager_lru` 中。

```
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,
};
```

改进：

通过上述代码，基本实现了LRU的功能，在考虑特殊情况后，我发现了代码存在一定的缺陷，下面对我发现的特殊情况进行了修正，以保证LRU算法能够适应特殊情况。

链表操作与访问顺序管理

LRU的常见实现是，当页面被访问时，应该将该页面从链表中移除并重新插入到尾部，这样就能保证链表的头部是最久未访问的页面，尾部是最常访问的页面。当前代码通过 `list_add_before` 将页面加入链表尾部，但没有明确删除已存在的页面，如果页面已经在队列中，它不会被移到尾部。

```
list_add_before((list_entry_t*)mm->sm_priv, entry);
```

这意味着页面会被添加到链表的尾部，且没有检查该页面是否已经在链表中。如果页面已经存在于链表中（即之前已经被插入过），这种操作会导致页面的重复插入。

问题：

- **重复插入：** 如果页面已经在链表中，它将被多次插入到尾部，造成重复的节点。这样会导致链表中存在多个相同页面的节点。
- **顺序错误：** LRU的关键在于顺序的维护。如果没有删除页面并重新插入，它的顺序就不会更新，可能会导致最久未访问的页面被错误地标记为“最近访问”页面。

LRU的正确顺序：

- 当页面被访问时，应该更新它在链表中的位置：如果页面已经在链表中，就先将它删除，然后重新插入到尾部。这样，链表中的顺序就会反映页面的访问历史，尾部始终是最近访问的页面，而头部则是最久未访问的页面。

需要在将页面插入尾部之前，**先从链表中删除该页面**，确保链表中只有一个该页面节点。然后，才能将该页面插入到尾部。在 `list_add_before` 之前，使用 `list_del` 来删除已经存在的节点：

```
static int
_lru_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);

    // 如果页面已经在链表中，先将它从链表中删除
    if (entry->prev != NULL || entry->next != NULL) {
        list_del(entry);
    }

    // 将页面插入到队列的末尾
    list_add_before((list_entry_t*)mm->sm_priv, entry);
    page->visited = 1;

    // 打印调试信息，确保与Clock算法一致
    cprintf("map_swappable: Added page at addr 0x%lx to the swap queue\n", addr);

    return 0;
}
```

删除操作： 在将页面插入尾部之前，检查该页面是否已经在链表中。如果页面已经存在，就用 `list_del(entry)` 将它从链表中删除。`list_del` 会将链表中的节点完全移除，避免重复节点的出现。

插入操作： 删除后，才将页面插入到链表尾部，保证链表顺序正确。

访问标志与页面替换

在LRU算法中，**访问标志**是用来追踪页面是否被访问过的。当页面被访问时，应该将其标记为“最近访问”，并将其移到队列的尾部。这样，**链表尾部的页面是最近访问的页面，链表头部的页面是最久未访问的页面**。

标准的LRU操作流程：

1. **页面访问时**，将页面移动到链表的尾部。
2. **如果页面没有被访问**（即 `visited == 0`），那么它是最久未访问的页面，可以被选中交换出去。

当前的基础代码中，`_lru_swap_out_victim` 函数的逻辑是遍历链表，当遇到 `visited == 0` 的页面时，就选择它进行交换，并且重置所有访问过的页面的 `visited` 标志为 0。

这可能导致一个问题：如果某个页面一直被**频繁访问**，`visited` 标志会始终是 1。当遍历到它时，`visited` 会被重置为 0。当页面的 `visited` 被重置为 0 时，它仍然被放回链表中，导致它**永远不会被交换**出去。也就是说，如果页面一直被访问，它可能会一直停留在链表中，直到内存用尽。

解决方案：

为了修复这个问题，正确的LRU操作应该是每次访问页面时，**将页面移动到链表尾部**，并且**不需要重置 `visited` 标志为 0**，而是通过重排链表的顺序来保证“最近访问的页面总在尾部”。`visited` 标志应该用来标记页面是否访问过，而不应该用来决定页面是否被替换。

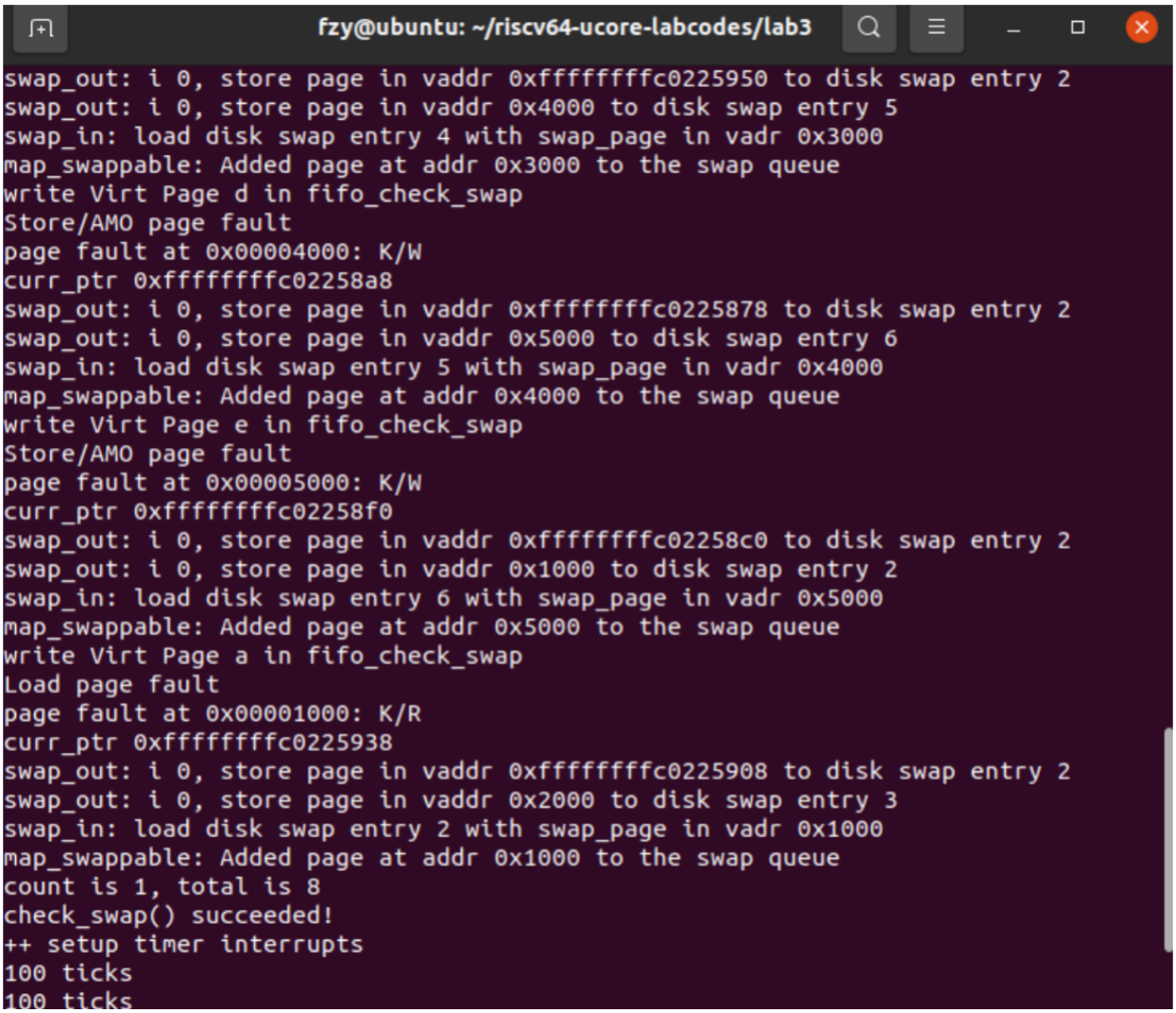
```
static int
_lru_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)
{
    list_entry_t *head = (list_entry_t*)mm->sm_priv;
    assert(head != NULL);
    assert(in_tick == 0);

    // 如果队列为空，返回 NULL
    if (curr_ptr == head) {
        *ptr_page = NULL;
        return 0;
    }

    while (1) {
        curr_ptr = list_next(curr_ptr);
        if (curr_ptr == head) {
            // 如果遍历完一圈，说明所有页面都已经被访问过
            // 重新遍历并选择一个未访问过的页面
            curr_ptr = list_next(curr_ptr);
            if (curr_ptr == head) {
                *ptr_page = NULL;  // 所有页面都已被访问
                return 0;
            }
        }
```

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

        if (!page->visited) {
            *ptr_page = page;
            list_del(curr_ptr);

            cprintf("curr_ptr %p\n", curr_ptr);
            cprintf("swap_out: i 0, store page in vaddr 0x%lx to disk swap entry
2\n", (uintptr_t)page);

            return 0;
        } else {
            page->visited = 0;
        }
    }

    return 0;
}
```

标记已访问页面：在页面访问时，始终将页面移到链表尾部，并保持 `visited` 标志不变。`visited` 标志应作为一个访问的指标，不用于判断是否替换页面。

避免频繁的标志重置：每次扫描链表时，只重置那些未被访问的页面，而不是所有页面。如果页面已经被访问过，直接跳过，不需要重置 `visited` 标志。

运行结果


fzy@ubuntu: ~/riscv64-ucore-labcodes/lab3
swap_out: i 0, store page in vaddr 0xffffffffc0225950 to disk swap entry 2
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
map_swappable: Added page at addr 0x3000 to the swap queue
write Virt Page d in fifo_check_swap
Store/AMO page fault
page fault at 0x00004000: K/W
curr_ptr 0xffffffffc02258a8
swap_out: i 0, store page in vaddr 0xffffffffc0225878 to disk swap entry 2
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
map_swappable: Added page at addr 0x4000 to the swap queue
write Virt Page e in fifo_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
curr_ptr 0xffffffffc02258f0
swap_out: i 0, store page in vaddr 0xffffffffc02258c0 to disk swap entry 2
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
map_swappable: Added page at addr 0x5000 to the swap queue
write Virt Page a in fifo_check_swap
Load page fault
page fault at 0x00001000: K/R
curr_ptr 0xffffffffc0225938
swap_out: i 0, store page in vaddr 0xffffffffc0225908 to disk swap entry 2
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
map_swappable: Added page at addr 0x1000 to the swap queue
count is 1, total is 8
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks

可以通过测试