LAB3

2112547张玉硕

2113099祝天智

2111288杜金轩

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？（为了方便同学们完成练习，所以实际上我们的项目代码和实验指导的还是略有不同，例如我们将FIFO页面置换算法头文件的大部分代码放在了`kern/mm/swap_fifo.c`文件中，这点请同学们注意）

- 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。如果你选择的函数不能完整地体现”从换入到换出“的过程，比如10个函数都是页面换入的时候调用的，或者解释功能的时候只解释了这10个函数在页面换入时的功能，那么也会扣除一定的分数

与页面替换有关的函数：

1

```
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

`_fifo_init_mm`函数用于初始化与FIFO页面置换算法相关的数据结构，为特定的`mm_struct`结构体（代表进程的内存管理信息）设置合适的数据

1. `list_init(&pra_list_head)`：使用`list_init`函数初始化一个名为`pra_list_head`的链表。这个链表将用于管理页面的置换顺序。在FIFO算法中，页面的置换顺序是按照它们进入内存的顺序进行的，链表用于维护页面的FIFO顺序。
2. `mm->sm_priv = &pra_list_head`：这一行代码将`mm`结构体中的`sm_priv`字段设置为`pra_list_head`的地址。`sm_priv`字段通常用于存储与页面置换算法相关的私有数据。在这里，它被设置为指向`pra_list_head`，使FIFO算法能够在置换页面时访问和维护链表。

2

```
_fifo_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);
    //record the page access situlation
    //(1)link the most recent arrival page at the back of the
pra_list_head qeueue.
    list_add(head, entry);
    return 0;
}
```

`_fifo_map_swappable` 函数，用于根据 FIFO 置换策略维护页面的置换顺序。

1. `struct mm_struct *mm`：传入的 `mm` 结构体表示了特定进程的内存管理信息。该函数会根据该进程的内存管理信息来执行页面置换。
2. `uintptr_t addr`：`addr` 是要操作的虚拟地址。函数会根据这个地址来操作相应的页面。
3. `struct Page *page`：这是一个指向 `Page` 结构体的指针，代表了要进行置换的页面。`Page` 结构体通常包含了页面的相关信息，如页面的物理地址、引用计数等。
4. `int swap_in`：这是一个标志，指示是否是页面置换进程。如果 `swap_in` 为非零值，表示是进行页面换入操作，否则是换出操作。

函数执行的操作：

- `list_entry_t *head = (list_entry_t *)mm->sm_priv;`：通过 `mm->sm_priv` 访问 FIFO 置换算法的私有数据，即前面所提到的链表（`pra_list_head`）的头部。
- `list_entry_t *entry = &(page->pra_page_link);`：获取要进行置换操作的页面的链接节点，这个节点包含在 `Page` 结构体的 `pra_page_link` 字段中。
- `list_add(head, entry);`：将要进行置换的页面的链接节点添加到链表的尾部。这是 FIFO 置换算法的关键步骤，它确保最近使用的页面在链表尾部，而最早使用的页面在链表头部，从而实现 FIFO 置换策略。

3

```
_fifo_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
    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;
}
```

`_fifo_swap_out_victim` 函数，用于选择并获取要进行页面置换的受害页面（victim）。

1. `struct mm_struct *mm`：传入的 `mm` 结构体表示了特定进程的内存管理信息。该函数会根据该进程的内存管理信息来执行页面置换。
2. `struct Page **ptr_page`：这是一个指向 `Page` 结构体指针的指针。函数将在 `ptr_page` 指向的地址存储选中的受害页面。
3. `int in_tick`：这是一个标志，指示是否是页面置换进程的时钟滴答。如果 `in_tick` 为非零值，表示是时钟滴答事件，否则不是。

函数执行的操作：

- `list_entry_t *head = (list_entry_t *)mm->sm_priv;`：通过 `mm->sm_priv` 访问 FIFO 置换算法的私有数据，即前面所提到的链表（`pra_list_head`）的头部。
- `assert(head != NULL);`：确保链表头部不为空。
- `assert(in_tick == 0);`：如果 `in_tick` 不是零，说明这是一个时钟滴答事件，不应该执行页面置换。
- `list_entry_t *entry = list_prev(head);`：获取链表中最早进入内存的页面。
- 如果 `entry` 不等于链表头部 `head`，表示链表不为空，可以执行页面置换。否则，将 `*ptr_page` 设置为 `NULL`，表示没有受害页面。
- 如果 `entry` 不为空，就执行以下操作：
    - `list_del(entry);`：将 `entry` 从链表中移除，即从 FIFO 队列中删除最早进入内存的页面。
    - `*ptr_page = le2page(entry, pra_page_link);`：将 `*ptr_page` 设置为 `entry` 对应的页面。这里使用了 `le2page` 宏，将链表节点转换为 `Page` 结构体，以获取页面信息。

最终，`_fifo_swap_out_victim` 函数选择并获取了符合 FIFO 置换策略的受害页面，并将其存储在 `ptr_page` 指向的地址。如果链表为空，表示没有可供置换的页面，将 `*ptr_page` 设置为 `NULL`。这个函数是 FIFO 置换算法中的一个关键组成部分，用于实现页面置换策略。

4

```
_fifo_check_swap(void)
```

`_fifo_check_swap` 函数用于检查 FIFO 页面置换算法的正确性和有效性。在这个函数中，它执行了一系列写入和读取操作，并验证了页面错误（page fault）的次数。

5

```
_fifo_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}
```

`_fifo_set_unswappable` 函数是 FIFO 页面置换算法中的一个辅助函数，用于将指定的虚拟地址标记为不可置换。在这个函数中，虽然它接受了一个虚拟地址 `addr` 和一个内存管理结构 `mm` 作为参数，但它实际上没有执行任何实际的标记或操作。它只是一个占位函数，通常情况下没有必要在 FIFO 页面置换算法中标记页面为不可置换，因此此函数返回 0，表示成功，但实际上并未执行任何操作。

6

```
struct Page *alloc_pages(size_t n) {
    struct Page *page = NULL;
    bool intr_flag;

    while (1) {
        local_intr_save(intr_flag);
        { page = pmm_manager->alloc_pages(n); }
        local_intr_restore(intr_flag);

        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        extern struct mm_struct *check_mm_struct;
        // cprintf("page %x, call swap_out in alloc_pages %d\n",page, n);
        swap_out(check_mm_struct, n, 0);
    }
    // cprintf("n %d,get page %x, No %d in alloc_pages\n",n,page,(page-pages));
    return page;
}
```

这段代码位于操作系统内存管理中，实现了页面的分配（`alloc_pages`）功能。

1. 首先，初始化一个指向 `struct Page` 结构的指针 `page`，初始值为 `NULL`，以及一个布尔变量 `intr_flag`。
2. 进入一个无限循环，表示要一直尝试分配页面，直到成功分配到页面为止。
3. 在循环内部，首先使用 `local_intr_save` 函数来保存中断状态，这是为了在执行分配页面操作期间禁用中断，以防止多个 CPU 同时访问相同的内存数据，确保分配过程是原子的。
4. 在禁用中断后，执行实际的页面分配操作，即调用 `pmm_manager->alloc_pages(n)` 函数来分配 `n` 个页面。这个具体的分配操作是由操作系统的内存管理模块实现的，`pmm_manager` 可能指向不同的内存管理器，如物理内存管理器（Physical Memory Manager）。
5. 接着，使用 `local_intr_restore` 函数来还原中断状态，即恢复之前保存的中断状态。

6. 然后，检查页面分配的结果。如果成功分配到页面，`page` 指针将指向分配到的页面，并跳出循环。
7. 如果无法分配页面，进一步检查 `n` 是否大于 1，以及 `swap_init_ok` 是否为 0。如果满足这两个条件，表示内存不足且虚拟内存交换机制已经初始化，可以使用虚拟内存页面置换操作 `swap_out` 来释放部分物理页面，以腾出空间。然后，程序重新进入下一轮循环，继续尝试页面分配。
8. 循环会一直尝试页面分配，直到成功分配到页面，然后将分配到的页面的指针 `page` 返回给调用者。

7

```
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 **ptr_page=NULL;
          struct Page *page;
          // cprintf("i %d, SWAP: call swap_out_victim\n",i);
          int r = sm->swap_out_victim(mm, &page, in_tick);
          if (r != 0) {
                    cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                   break;
          }
          //assert(!PageReserved(page));

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

          v=page->pra_vaddr;
          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) {
                    cprintf("SWAP: failed to save\n");
                    sm->map_swappable(mm, v, page, 0);
                    continue;
          }
          else {
                    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);
          }

          tlb_invalidate(mm->pgdir, v);
     }
     return i;
```

```
38  }
```

这段代码位于操作系统内存管理模块，实现了将页面换出到磁盘的逻辑，

1. 循环 `n` 次，表示要将 `n` 个页面换出到磁盘。
2. 对于每次循环迭代，首先声明一个名为 `v` 的 `uintptr_t` 变量，用于存储将要被换出的页面的虚拟地址。
3. 声明 `struct Page` 类型的指针 `page`，这个指针将在后面的代码中用于存储被选中的要换出的页面。
4. 调用 `sm->swap_out_victim` 函数，尝试选取一个要被换出的页面。如果成功选取页面，它会在 `page` 中存储该页面的信息。
5. 如果 `sm->swap_out_victim` 函数返回非零值，表示选取页面失败，打印错误信息并跳出循环。
6. 获取被选中页面的虚拟地址 `v`。
7. 调用 `get_pte` 函数获取虚拟地址 `v` 对应的页表项（Page Table Entry），将其存储在 `ptep` 中。
8. 断言确保页表项有效（`PTE_V` 标志位被设置）。
9. 调用 `swapfs_write` 函数，将选中的页面写入磁盘的交换分区中。交换分区的偏移量由 `(page->pra_vaddr/PGSIZE + 1) << 8` 计算得出，这个偏移量通常会存储在页表项中，表示页面在磁盘中的位置。
10. 如果 `swapfs_write` 函数返回非零值，表示写入磁盘失败。此时，需要调用 `sm->map_swappable` 函数将页面重新标记为可置换的，然后继续下一次迭代。
11. 如果写入磁盘成功，打印信息表示成功将页面存储到磁盘中，然后更新页表项 `ptep` 的内容，将页面的标识（`(page->pra_vaddr/PGSIZE + 1) << 8`）存储在页表项中，表示页面已经在磁盘中，之后释放页面 `page`。
12. 最后，调用 `tlb_invalidate` 函数，将虚拟地址 `v` 对应的 TLB（翻译后备缓存）项失效，以确保之后的访问将会触发重新加载页表项。

8

```
1  int
2  swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
3  {
4       struct Page *result = alloc_page();
5       assert(result!=NULL);
6
7       pte_t *ptep = get_pte(mm->pgdir, addr, 0);
8       // cprintf("SWAP: load ptep %x swap entry %d to vaddr 0x%08x, page
   %x, No %d\n", ptep, (*ptep)>>8, addr, result, (result-pages));
9
10      int r;
11      if ((r = swapfs_read((*ptep), result)) != 0)
12      {
13         assert(r!=0);
14      }
15      cprintf("swap_in: load disk swap entry %d with swap_page in vadr
   0x%x\n", (*ptep)>>8, addr);
16      *ptr_result=result;
17      return 0;
```

```
}
```

这段代码位于操作系统内存管理模块，实现了将页面从磁盘加载到物理内存的逻辑

1. 首先分配一个名为 `result` 的物理页面，该页面用于存储从磁盘加载的数据。
2. 使用断言 `assert` 来确保分配页面成功，如果失败，会触发断言错误。
3. 调用 `get_pte` 函数，获取虚拟地址 `addr` 对应的页表项 `ptep`，这是为了获得交换分区中的页表项，其中包含了磁盘中页面的位置信息。
4. 调用 `swapfs_read` 函数，将磁盘上的数据读取到刚刚分配的 `result` 页面中。读取的位置由页表项 `ptep` 中的值确定。
5. 如果 `swapfs_read` 函数返回非零值，表示读取磁盘失败，会触发断言错误。
6. 最后，打印信息表示成功从磁盘加载页面到物理内存中，将 `result` 页面的地址存储在 `ptr_result` 指针中，以便返回给调用者。

9

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /*

    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);
    }
    pde_t *pdep0 = &((pde_t *)KADDR(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);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

这段代码实现了根据虚拟地址 `la` 获取或创建页表项（Page Table Entry，PTE）的功能。这个函数的作用是实现虚拟地址到物理地址的映射。以下是代码的主要逻辑和注释：

1. 首先，声明一个指向页表目录项（Page Directory Entry，PDE）的指针 `pdep1`，用于存储虚拟地址 `la` 对应的一级页表目录项。
2. 判断这个页表目录项是否有效，即是否已存在。如果无效，需要创建。

3. 如果 `create` 参数为 `false`，表示不需要创建页表项，直接返回 `NULL` 表示失败。
4. 如果需要创建页表项或者页表目录项已经存在，执行以下操作：
    - 分配一个物理页面（`page`）用于存储数据。
    - 设置该页面的引用计数为 1（表示有一个引用），并获得该物理页面的物理地址（`pa`）。
    - 使用 `memset` 函数将该物理页面清零。
    - 使用 `pte_create` 宏创建页表项，并将这个页表项存储到页表目录项 `pdep1` 中。
5. 接下来，声明一个指向二级页表目录项（PDE）的指针 `pdep0`，用于存储虚拟地址 `la` 对应的二级页表目录项。
6. 类似地，判断二级页表目录项是否有效，如果无效，需要创建。
7. 如果需要创建页表项或者页表目录项已经存在，执行以下操作：
    - 再次分配一个物理页面（`page`）用于存储数据。
    - 设置该页面的引用计数为 1（表示有一个引用），并获得该物理页面的物理地址（`pa`）。
    - 使用 `memset` 函数将该物理页面清零。
    - 使用 `pte_create` 宏创建页表项，并将这个页表项存储到页表目录项 `pdep0` 中。
8. 最后，通过指针操作，返回虚拟地址 `la` 对应的页表项的指针。

10

```
struct Page *get_page(pde_t *pgdir, uintptr_t la, pte_t **ptep_store) {
    pte_t *ptep = get_pte(pgdir, la, 0);
    if (ptep_store != NULL) {
        *ptep_store = ptep;
    }
    if (ptep != NULL && *ptep & PTE_V) {
        return pte2page(*ptep);
    }
    return NULL;
}
```

这段代码定义了一个函数 `get_page`，其目的是根据给定的页表 `pgdir` 和虚拟地址 `la`，获取对应的物理页结构指针（`struct Page*`）并可选地返回页表项指针（`pte_t**`）。

1. 调用 `get_pte` 函数，传递给它页表 `pgdir` 和虚拟地址 `la`，并将 `create` 参数设为 0（不创建页表项）来获取对应虚拟地址的页表项指针 `ptep`。
2. 如果传递给 `get_page` 函数的参数 `ptep_store` 不为 `NULL`，则将 `ptep` 的值赋给 `*ptep_store`，这样调用者可以获取到页表项的指针，用于后续操作。
3. 检查 `ptep` 是否为 `NULL`，如果为 `NULL`，表示没有找到页表项，或者页表项无效。此时返回 `NULL`，表示没有找到对应的物理页结构。
4. 如果 `ptep` 不为 `NULL` 并且页表项中的 PTE_V 标志为 1（有效），则表示找到了一个有效的页表项。使用 `pte2page` 宏从页表项中提取出对应的物理页结构指针，然后将其返回。

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？

这函数是一个用于管理虚拟地址到物理地址映射的关键函数。 sv32、sv39、sv48 中的页表结构的基本原则是相同的，只有一些参数的值不同：

1. **sv32（32位）：** 采用两级页表结构，分为页目录表（Page Directory Table，PDT）和页表（Page Table，PT）两级。页目录表项（PDE）和页表项（PTE）是 32 位。虚拟地址的高 10 位（31-22）用于查找页目录表项，接下来的 10 位（21-12）用于查找页表项。
2. **sv39（39位）：** 采用三级页表结构，分为页全局目录表（Page Global Directory Table，PGDT）、页目录表（PDT）、和页表（PT）三级。PGDT 表项和PDT 和 PT 表项是 64 位。虚拟地址的高 9 位（38-30）用于查找 PGDT 表项，接下来的 9 位（29-21）用于查找 PDT 表项，最后的 9 位（20-12）用于查找 PT 表项。
3. **sv48（48位）：** 采用四级页表结构，分为页全局目录指针表（Page Global Directory Pointer Table，PGDPT）、页全局目录表（PGDT）、页目录表（PDT），和页表（PT）四级。PGDPT 表项是 64 位，PGDT、PDT 和 PT 表项是 64 位。虚拟地址的高 9 位（47-39）用于查找 PGDPT 表项，接下来的 9 位（38-30）用于查找 PGDT 表项，再接下来的 9 位（29-21）用于查找 PDT 表项，最后的 9 位（20-12）用于查找 PT 表项。
4. 这两段相似的代码之所以如此相像，是因为它们实现了一样的功能，但针对不同位数的页表结构（sv32、sv39、sv48）。因此，它们都遵循相同的基本逻辑，只有位数和特定参数有所不同。通过参数的传递，可以在不同的页表结构下复用相同的代码。
5. 将页表项的查找和分配合并在一个函数中是一种通用的设计，允许在需要查找页表项的同时，如果它不存在，也可以立即分配并初始化它。这种设计减少了冗余代码，使代码更加清晰。分离这两个功能可能会引入更多的复杂性，需要多次扫描页表结构，从而降低了代码的效率。因此，将它们合并在一个函数中是合理的，而不会带来额外的复杂性。

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。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。
- 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?
- 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?

（1）

以下说明中将页目录项和页表项统称为页表项。Sv39 下一个页表项的组成如下：

```
+---+----+---------+-------+-------+-------+---+---+---+---+---+---+---+---+---+
| N |PBMT| Reserved| PPN[2]| PPN[1]| PPN[0]|RSW| D | A | G | U | X | W | R | V |
+---+----+---------+-------+-------+-------+---+---+---+---+---+---+---+---+---+
 1   2    7    26    9    9    2  1   1   1   1   1   1   1   1
```

低八位用于表示页表项的属性和权限。其中 V 标志位表示整个页表项是否合法，若 V 为 0，则整个页表项中的其余比特均无意义，并且可以由软件自由定义使用方式。

ucore 中页替换算法利用了这个 RISC-V 特权级的约定。当一个页面被换出时，页表项位置的值会被如下设置：

*ptep = (page->pra_vaddr / PGSIZE + 1) << 8;

这会将对应页的虚拟页号加一并左移，保证其低八位为 0，并且整个值不为 0（表示是换出而不是被新建）。在 ucore 的实现中，一个被换出的页面中存储的内容约定为一个 swap_entry_t 类型的值，这个类型的值需要保证最低位即 V 标志位为 0，从而保证硬件访问这个页表项时会产生缺页异常。

通过在页表项中的标记，可以在 do_pgfault 时正确决定是需要为一个地址重新分配页并且建立映射关系还是从“硬盘”中换入页面。

在 sv39 中，页表项中的一些标记位和保留位（RSW）可以在页替换算法中用于实现一些功能。例如，A (Accessed) 可以表示一个页是否被访问（写或读）过，D (Dirty) 表示页是否被修改（写入）过，这两个标记位可以用于实现扩展的 Clock 页面置换算法。

（2）

当出现页访问异常时，硬件需要根据发生异常的类型设置 scause 寄存器，将产生异常的指令地址存入 sepc 寄存器，将访问的地址存入 stval，并且根据设置的 stvec 进入操作系统的异常处理过程。之后再次调用操作系统的缺页服务例程进行缺页处理，从而陷入死循环。

（3）

Sv39 分页机制下，每一个页表所占用的空间刚好为一个页的大小。在处理缺页时，如果一个虚拟地址对应的二级、三级页表项（页目录项）不存在，则会为其分配一个页，当第一级页表项没有设置过时也会分配一个页。此外，当一个页面被换出时，他所对应的页面会被释放，当一个页面被换入或者新建时，会分配一个页面。所以，对于本实验中缺页机制所处理和分配的所有页目录项、页表项，都对应于 pages 数组中的一个页，但是 pages 中的页并不一定会全部使用。

1. **页目录项和页表项的组成部分对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）包含了有关虚拟内存管理的重要信息。这些信息对于ucore实现页替换算法（如LRU或Clock算法）非常重要。以下是它们的一些潜在用处：
        - **存在位（Present Bit）**：PTE和PDE中的存在位指示该页是否在物理内存中。在页替换算法中，它可以用来确定哪些页在内存中，哪些页不在内存中。
        - **访问位（Accessed Bit）**：用于指示一个页面是否被访问过。页替换算法可以使用它来判断哪些页面最近被访问过，从而做出替换决策。

        - **脏位（Dirty Bit）**：脏位表示一个页面是否已经被修改。页替换算法可以使用它来判断哪些页面需要写回到磁盘，从而避免数据丢失。
        - **读/写权限位（Read/Write Permissions）**：指示一个页面是否可读、可写或只读。页替换算法可能需要根据权限位来决定哪些页面可以被替换或保持在内存中。
        - **物理地址指针（Physical Address Pointer）**：PTE中通常包含指向物理页框的指针。这对于页替换算法来说非常重要，因为它告诉算法哪些物理页是被占用的，哪些是空闲的。
2. **硬件在缺页服务例程中的操作**：当ucore的缺页服务例程在执行过程中出现页访问异常，硬件通常会执行以下操作：
    - **保存上下文**：硬件会保存当前进程的上下文（寄存器状态、程序计数器等）。
    - **触发异常**：硬件会生成一个异常，通常是页访问异常，以通知操作系统发生了一个错误。
    - **跳转到异常处理程序**：硬件会跳转到操作系统内核中的页异常处理程序。
    - **处理异常**：操作系统的页异常处理程序会分析异常的原因，通常是缺页异常，然后执行以下操作：
        - 从硬盘加载缺失的页面到内存中（如果页面在磁盘上）。
        - 更新页表，将虚拟地址映射到新加载的页面。
        - 恢复进程的上下文，以便它可以继续执行。
    - **继续执行**：一旦异常处理程序完成，硬件会返回到进程的上下文，使其继续执行。
3. **数据结构Page的全局变量与页表中的页目录项和页表项的对应关系**：
    - 在ucore中，数据结构Page的全局变量通常用于跟踪物理页面的状态和信息。每个Page结构表示一个物理页面，而与之对应的是页表中的页表项（PTE）。
    - 每个Page结构通常包含以下信息：
        - 物理页面的状态（例如，是否被占用、脏位等）。
        - 物理页面的地址（物理地址）。
        - 指向下一个Page结构的指针，用于形成一个链表，以跟踪空闲的物理页面。
    - 页表中的页表项（PTE）包含了与Page结构相关的信息，例如物理页面的地址。
    - 这两者之间的对应关系在ucore中可以通过数据结构Page的指针来建立。当需要查找特定物理页面的信息时，可以通过Page结构的指针找到对应的PTE，反之亦然。这种对应关系有助于操作系统管理物理页面并与虚拟地址空间建立正确的映射。这种对应关系的建立和维护对于页表的操作和页替换算法非常重要。

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。

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。

实现方法：

（1）为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一个循环队列

（2）当某页被访问时，其访问位置为1

（3）当需要淘汰一个页面时，只需检查页的访问位：如果是0，选择此页换出；如果是1，将它置0，暂不换出，继续检查下一个页面

（4）若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0，再进行第二轮扫描，第二轮扫描中一定会有访问位为0的页面，因此简单的CLOCK算法选择一个淘汰页面最多会经过两轮扫描

```
static int
_clock_init_mm(struct mm_struct *mm)//初始化
{
     list_init(&pra_list_head);
     curr_ptr=&pra_list_head;
     mm->sm_priv = &pra_list_head;
     /*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
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

`_clock_init_mm`函数用于初始化与CLOCK页面置换算法相关的数据结构，为特定的`mm_struct`结构体（代表进程的内存管理信息）设置合适的数据

1. `list_init(&pra_list_head);`：初始化一个双向链表`pra_list_head`为空链表。这个链表将用于存储页面替换算法的替换候选页。
2. `curr_ptr=&pra_list_head;`：初始化当前指针`curr_ptr`，使其指向链表的头部（即`pra_list_head`）。这个指针用于表示当前页面替换位置，初始化为链表头。
3. `mm->sm_priv = &pra_list_head;`：将`mm`结构的私有成员指针`sm_priv`指向`pra_list_head`。这样，`mm`结构就可以访问`pra_list_head`，以便后续的页面替换算法可以操作该链表。

```
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
{
    cprintf("clock swappable done!");

    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && curr_ptr!= NULL);
    //record the page access situlation
    //(1)link the most recent arrival page at the back of the
pra_list_head qeueue.
    list_add(head, entry);//如果本来存在，则移到最后
    // struct Page *p = le2page(entry, pra_page_link);
    page->visited=1;//设置标记
    return 0;
}
```

`_clock_map_swappable`函数用于将一个页面映射为可交换的，通常是在将页面从磁盘加载到内存时调用。

1. `list_entry_t *head = (list_entry_t*)mm->sm_priv;`：从`mm`的私有成员中获取链表中的替换候选页。
2. `list_entry_t *entry = &(page->pra_page_link);`：获取将要映射为可交换的页面的链表入口。
3. `list_add(head, entry);`：将`entry`添加到`head`（链表的尾部），这表示将当前页面放到链表的末尾，表示它是最近使用的页面。
4. `page->visited = 1;`：设置页面的`visited`标志为1，表示该页面已被访问过。

```
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page,
int in_tick)
{
    cprintf("clock swapout done!");
    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
    //遍历找到第一个没有被标记为最近使用的页，并且把标记为最近使用的页的标记
设置为最近未使用
    while (1) {

        cprintf("loop");
        list_entry_t* entry = list_prev(head);//从head找到链表第一个
        struct Page *p = le2page(entry, pra_page_link);
        if (entry == head) {
```

```
            *ptr_page = NULL;
            break;
        }
        if(p->visited==0)// 为0挑出来
        {
            list_del(entry);
            *ptr_page = le2page(entry, pra_page_link);
            cprintf("curr_ptr %p\n", curr_ptr);
            break;

        }
        if(p->visited==1)//为1置为0
        {
            p->visited=0;
            curr_ptr = entry;
            entry=list_prev(entry);
        }
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 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
    }
    return 0;
}
```

`_clock_swap_out_victim`函数用于选择要被换出的牺牲页面（victim）。

1. `list_entry_t *head = (list_entry_t*)mm->sm_priv;`：从`mm`的私有成员中获取用于页面替换算法的链表的头部指针。
2. `if (p->visited == 0)`：遍历链表找到，如果页面的`visited`标志为0，表示这是一个未被访问的页面，将其从链表中删除，并将其指针赋给`ptr_page`，作为要被换出的页面。
3. `if (p->visited == 1)`：如果页面的`visited`标志为1，表示这是一个最近访问过的页面，将其`visited`标志设置为0，表示已经重新访问过。
4. 如果都被访问过，那么就会遍历第二遍。

改进型的CLOCK置换算法

1. 引入：简单的时钟置换算法仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过，就不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。
2. 思想：因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进型的时钟置换算法的思想。
3. 实现方法：修改位=0，表示页面没有被修改过；修改位=1，表示页面被修改过。为方便讨论，用（访问位，修改位）的形式表示各页面状态。

Clock页替换算法和FIFO算法的不同之处：

1. **页面替换策略**：
   - **FIFO算法**：FIFO算法基于最早进入内存的页面被置换出去的原则。它维护一个页面队列，新的页面加入队列末尾，而需要替换页面时，选择队列头部的页面。
   - **Clock算法**：Clock算法是一种改进的FIFO算法。它也维护一个页面队列，但不一定选择队列头部的页面，而是通过检查页面的"访问位"（或称为"使用位"）来确定是否页面最近被使用。Clock算法会以循环方式检查页面队列，类似于时钟的指针，找到第一个未被访问的页面进行替换。
2. **页面置换效率**：
   - **FIFO算法**：FIFO算法可能会导致"Belady's Anomaly"，即在增加内存时，缺页次数反而增加。因为FIFO只关注页面的进入顺序，而不考虑页面是否频繁使用。
   - **Clock算法**：Clock算法通过访问位考虑了页面的使用情况，因此在某些情况下可能更高效，因为它会尽量保留那些频繁使用的页面。
3. **实现复杂性**：
   - **FIFO算法**：FIFO算法非常简单，只需要维护一个队列，因此实现相对容易。
   - **Clock算法**：Clock算法相对复杂一些，因为它需要维护一个额外的位（访问位），并定期检查这个位，所以实现上可能稍显复杂。
4. **替换精度**：
   - **FIFO算法**：FIFO算法没有考虑页面的使用情况，只根据页面进入内存的时间，因此可能导致一些频繁使用的页面被置换出去。
   - **Clock算法**：Clock算法尝试更准确地估计页面的使用情况，因此在某些情况下能更好地适应程序的访问模式。

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？

优势：

1. **减少页表项数量：** 采用大页的方式可以减少页表的层次结构，大页可以映射更多的物理内存。降低了页表项的数量，减少了页表维护的开销，提高了内存访问的效率。
2. **减少TLB缺失：** 大页可以更好地利用翻译后备缓冲（TLB），减少了TLB缺失的次数。因为更多的虚拟地址可以映射到同一个TLB条目中，更多的内存访问可以从TLB中快速获得物理地址。
3. **提高性能：** 减少页表项和TLB缺失从而达到更快的内存访问速度，提高了应用程序的性能。

劣势：

1. **内部碎片：** 大页可能导致内部碎片，因为如果一个大页没有完全填满，剩余的部分会浪费掉。这可能浪费了物理内存。
2. **不适用于所有应用：** 大页不适用于所有应用程序。某些应用程序可能需要更细粒度的内存管理，而大页可能无法满足它们的需求。
3. **不适用于小内存系统：** 大页需要更多的物理内存来支持，因此可能不适用于小内存系统，因为大页可能导致物理内存不足。

4. **不灵活：** 大页可能不够灵活，因为它们的大小是固定的，无法根据应用程序的实际需求进行调整。

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。

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（最近最少使用）页面置换算法是根据过去使用次数来预测将来的页面使用情况，具体过程如下：

当发生缺页而分配给进程的物理块耗尽时，需要将所需页面置换入内存，被换出的页面是最近最少被使用的页面。具体的实现方法为为每个内存中的页面配置一个移位寄存器，每过一段时间（如10ms）将其中的数字右移。当该页面被访问时，将最高位置一，这样，最近被访问的页面寄存器中的数字是最大的，需要换出页面时，只需选取数值最小的页面换出；另一种方法是为每个进程配置一个栈，页面进入内存时将其压入栈中，当访问内存中的页面时，将其重新压入栈顶，如此最近被访问的页面就在栈顶，需要置换页面时，只需将栈底的页面置换出即可。

```
list_entry_t pra_list_head;
static int
_lru_init_mm(struct mm_struct *mm)
{
    // 初始化
    list_init(&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;
    return 0;
}
```

`_lru_init_mm`初始化 LRU（最近最少使用）页面替换算法所需的数据结构和指针。`_lru_init_mm` 函数是初始化进程内存管理结构 `mm` 的函数。在该函数中，它完成以下任务：

1.使用 `list_init(&pra_list_head)` 初始化 `pra_list_head` 链表头，将其设置为空链表。

2.将 `mm` 的 `sm_priv` 字段指针设置为 `&pra_list_head`。这样，`mm` 结构中的 `sm_priv` 字段将指向 `pra_list_head` 链表头，以便后续的页面替换算法可以访问和操作这个链表。

```
static int
_lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in)
{
    cprintf("lru swappable done!");

    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(head, entry);//如果本来存在，则移到最后

    int num_bits = sizeof(uint_t) * 8;// 32位系统
    uint_t mask = (uint_t)1 << (num_bits - 1);// 将最高位设置为1

    page->visited |= mask;  // 以访问便按位或，将最高位设置为1
    return 0;

}

```

`_lru_map_swappable`函数用于 LRU页面替换算法的页面映射函数，大体与fifo算法部分类似，不同的是，会设置一个visited标志，将其最高位设置为1.

1. `list_entry_t *head=(list_entry_t*) mm->sm_priv;` 这行代码从 `mm` 的 `sm_priv` 字段中获取指向页面链表头的指针，并将其存储在 `head` 变量中。
2. `list_entry_t *entry=&(page->pra_page_link);` 这行代码从输入的 `page` 结构中获取 `pra_page_link` 字段的指针，并将其存储在 `entry` 变量中。这表示要将当前页面添加到链表中。
3. `list_add(head, entry);` 这行代码将 `entry` 添加到 `head` 链表的末尾。如果页面已经存在于链表中，它将被移到链表的末尾。这样，链表将按照页面的访问顺序进行排列。
4. `int num_bits = sizeof(uint_t) * 8;` 这行代码计算 `uint_t` 类型的位数，通常是32位系统，所以 `num_bits` 的值为32。
5. `uint_t mask = (uint_t)1 << (num_bits - 1);` 这行代码创建一个掩码 `mask`，将最高位设置为1。这是为了用于表示页面的访问情况，将最高位设置为1表示页面最近被访问过。
6. `page->visited |= mask;` 这行代码将 `page` 结构中的 `visited` 字段与 `mask` 执行按位或操作，将最高位设置为1，表示页面最近被访问过。

```
#include <defs.h>
#include <riscv.h>
#include <stdio.h>
#include <string.h>
#include <swap.h>
#include <swap_lru.h>


#include <list.h>

list_entry_t pra_list_head;
static int
_lru_init_mm(struct mm_struct *mm)
{
    // 初始化
    list_init(&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;
    return 0;
```

```
}



static int
_lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
{
    cprintf("lru swappable done!");

    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(head, entry);//如果本来存在，则移到最后

    int num_bits = sizeof(uint_t) * 8;// 32位系统
    uint_t mask = (uint_t)1 << (num_bits - 1);// 将最高位设置为1

    page->visited |= mask;  // 以访问便按位或，将最高位设置为1
    return 0;

}


static int
_lru_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)
{
    list_entry_t *entry = list_next(&pra_list_head);
    struct Page *victim = NULL;

    // 如果链表为空，没有可供置换的页面
    if (entry == &pra_list_head) {
        *ptr_page = NULL;
        return 0;
    }

    uint_t min_visited = 4294967295;
    while (entry != &pra_list_head) {
        struct Page *page = le2page(entry, pra_page_link);
        if (page->visited <= min_visited) {
            victim = page;
            min_visited = page->visited;
        }
        entry = list_next(entry);
    }

    if (victim == NULL) {
        entry = list_next(&pra_list_head);
```

```
        victim = le2page(entry, pra_page_link);
    }
    // 从链表中删除选中的页面
    list_del(entry);
    // 将选中的页面返回给 caller
    *ptr_page = victim;
    return 0;
}

static int
_lru_check_swap(void) {
#ifdef ucore_test
    int score = 0, totalscore = 5;
    cprintf("%d\n", &score);
    ++ score; cprintf("grading %d/%d points", score, totalscore);
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==4);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==4);
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==4);
    *(unsigned char *)0x2000 = 0x0b;
    ++ score; cprintf("grading %d/%d points", score, totalscore);
    assert(pgfault_num==4);
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    ++ score; cprintf("grading %d/%d points", score, totalscore);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==5);
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==5);
    ++ score; cprintf("grading %d/%d points", score, totalscore);
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==5);
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==6);
    ++ score; cprintf("grading %d/%d points", score, totalscore);
#else
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==4);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==4);
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==4);
```

```
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==4);
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==5);
    *(unsigned char *)0x2000 = 0x0b;
    assert(pgfault_num==5);
    *(unsigned char *)0x3000 = 0x0c;
    assert(pgfault_num==5);
    *(unsigned char *)0x4000 = 0x0d;
    assert(pgfault_num==5);
    *(unsigned char *)0x5000 = 0x0e;
    assert(pgfault_num==5);
    assert(*(unsigned char *)0x1000 == 0x0a);
    *(unsigned char *)0x1000 = 0x0a;
    assert(pgfault_num==6);

#endif
    return 0;
}


static int
_lru_init(void)
{
    return 0;
}

static int
_lru_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
    return 0;
}

static int
_lru_tick_event(struct mm_struct *mm)
{ return 0; }


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,
};
```

`_lru_swap_out_victim`函数用于选择需要置换出的页面：

1. `list_entry_t *entry = list_next(&pra_list_head);` 这行代码从页面链表的头部（`pra_list_head`）获取第一个页面的指针（`entry`）。
2. `struct Page *victim = NULL;` 这行代码初始化 `victim` 指针，表示当前没有选中的受害者页面。
3. 接下来，代码进入一个循环，遍历页面链表，查找最适合作为受害者的页面。`if (page->visited <= min_visited)` 这行代码比较当前页面的访问情况（`visited` 字段）,找到最进最少使用的页面。如果是，就更新 `max_visited` 为当前页面的访问情况，同时将 `victim` 指向当前页面。这里我们是通过每次方法页面时，将页面的`visited`标志位的最高位置为1，同时每隔一定时钟周期，我们会将链表中的每个页面的`visited`标志右移一位，所有越大代表最近使用越多。
4. 最后，将选中的页面指针 `victim` 存储到 `ptr_page` 中，以便后续的置换操作。

```
static uint64_t last_trigger_time = 0;
static uint64_t trigger_interval = 5; // 触发间隔不知道设置为多少

int
swap_tick_event(struct mm_struct *mm)
{// 每隔一段时间将所有visited标志右移一位。
     if(sm->name == "lru swap manager"){// 如果是lru算法
          uint64_t current_time = get_current_time(); // 获取当前时间

          if (current_time - last_trigger_time >= trigger_interval) {
               // 检查是否满足触发条件
               // 执行需要触发的操作
               list_entry_t *head=(list_entry_t*) mm->sm_priv;
               list_entry_t *cur;
               if(head != NULL){
                    cur = list_next(head);
                    while (cur != head) {
                         struct Page *cur_page = le2page(cur,
pra_page_link);
                         cur_page->visited = cur_page->visited >> 1;
                         cur = list_next(cur);
                    }
               }
               // 更新上次触发时间
               last_trigger_time = current_time;
          }
     }
     return sm->tick_event(mm);
}
```

`swap_tick_event`函数用于每隔一定时钟周期，对所有页面的标志位进行右移。