南开大学操作系统实验报告

学号：2110598 姓名：许宸

学号：2113384 姓名：刘新宇

学号：2112487 姓名：刘轩宇

实验题目 — Lab 3 缺页异常和页面置换

实验目的

- 了解虚拟内存的 Page Fault 异常处理实现
- 了解页替换算法在操作系统中的实现
- 学会如何使用多级页表，处理缺页异常（Page Fault），实现页面置换算法

环境

软件环境

ubuntu22.04，QEMU-4.1.1

实验步骤与内容

（一）练习 1：理解基于 FIFO 的页面替换算法

在FIFO页面置换算法中，以下函数/宏对从页面换入到页面换出的过程有实际影响：

1. `swap_init()`:在Init.c的kern_init()内核初始化函数中进行swap的全局初始化，在本初始化函数中进行了交换文件系统的初始化(swapfs_init()函数)，对交换最大偏移量的合法性进行了检查并指定了页面交换算法。

2. `do_pgfault()`:当一个页面故障（page fault）发生时，内核会调用这个函数来处理页面故障异常。

3. `find_vma`:这个函数用于查找给定地址所属的虚拟内存区域（VMA）。它会遍历一个进程的 VMA 列表，查找包含给定地址的 VMA，并返回对应的 VMA 结构。

4. `PTE_U`:这是一个宏，用于表示页面表项（Page Table Entry）中的用户权限位。如果该位被设置，表示用户可以访问该页，否则不可以。

5. `swap_in()`:这个函数用于将页面从磁盘交换回内存。它接受一个内存管理结构、地址和页面的指针，并根据地址和管理结构从磁盘中加载页面内容，然后建立页面和地址的映射。

6. `page_insert()`:这个函数用于将页面插入到页表中，建立虚拟地址到物理地址的映射。通常在页面缺失（page fault）处理中使用。

7. `_fifo_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)`: 将页面添加到FIFO页面队列的末尾，表示最近被访问。

8. `get_pte()`:这个函数用于获取页表项（PTE），通过页目录表和给定的线性地址。它还提供了一个选项，可以选择是否在页表项不存在时创建一个新的页表项。

9. `pte_create()`:这是一个宏，用于创建页表项，设置页面的物理地址和权限位。

10. `PDX1`:这是一个宏，用于获取虚拟地址的一级页目录项的索引。通常用于页表的索引计算。

11. `KADDR`:这是一个宏，用于将物理地址转换为内核虚拟地址。通常用于直接访问内核中的数据结构。

12. `page2pa`:这是一个宏，用于获取 struct Page 结构所管理的页面的物理地址。通常在内核中需要获取页面的物理地址时使用。

（二）练习 2：深入理解不同分页模式的工作原理

`get_pte()` 函数的两段相似代码用于处理不同级别的页表项。在32位虚拟地址模式下（sv32），处理两级页表；在64位虚拟地址模式下（sv39或sv48），处理三级或四级页表。这两段代码相似的原因是因为页表的结构和访问方式在不同级别的页表中基本相同，只是位数和级别不同。
https://junimay.github.io/wiki/RISCV/%E5%88%86%E9%A1%B5%E6%9C%BA%E5%88%B6/

虽然这两段代码有相似之处，但它们也有一些差异，如页表的级别和页表项的位数，以及页表项的标志位。因此，在处理两级页表和三级/四级页表时，需要分别处理这些不同之处，从而保证代码的正确性。

关于将页表项的查找和分配合并在一个函数中的写法，是否好取决于代码的设计和需求。将它们合并在一个函数中有一些优点：

1. **代码结构简单：** 合并在一个函数中可以减少代码重复，提高代码的可维护性。避免了在多个函数之间传递相关参数。

2. **减少函数调用：** 将查找和分配分开可能会导致多个函数之间的调用关系，增加了函数调用开销。合并在一个函数中可以减少这种开销。

3. **一致性：** 代码中的逻辑一致性更容易维护，因为查找和分配是紧密相关的操作。

然而，是否将它们分开取决于项目的需求。如果项目中更注重模块化和代码分离，或者需要在不同情况下对它们进行不同的处理，那么将它们分开可能更有意义。拆分功能可以提高代码的可读性和可维护性，但也会增加代码量。

（三）练习 3：给未被映射的地址映射上物理页

```
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 翻译：尝试找到一个包含addr的vma
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++; // 页错误数加一
    //If the addr is in the range of a mm's vma? 翻译：如果addr在mm的vma的范围内?
    if (vma == NULL || vma->vm_start > addr) { // 如果vma为空或者vma的起始地址大于
addr
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
```

```
    uint32_t perm = PTE_U; // PTE_U: page table/directory entry flags bit : User
can access   pern: 权限
    if (vma->vm_flags & VM_WRITE) { // 如果vma的标志位中包含VM_WRITE
        perm |= (PTE_R | PTE_W); // PTE_R: page table/directory entry flags bit :
Readable
                                 // PTE_W: page table/directory entry flags bit :
Writeable
    }
    addr = ROUNDDOWN(addr, PGSIZE); // ROUNDDOWN: 向下取整

    ret = -E_NO_MEM; // E_NO_MEM: Request failed due to memory shortage 翻译: 由于
内存短缺而请求失败

    pte_t *ptep=NULL; // ptep: 页表项指针

    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;
            // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
            //(1) According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in failed with error code %d\n", ret);
                goto failed;
            }
            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            page_insert(mm->pgdir, page, addr, perm);
            //(3) make the page swappable.
            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;
}
```

（四）练习 4：补充完成 Clock 页替换算法

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

list_entry_t pra_list_head, *curr_ptr;

static int
_clock_init_mm(struct mm_struct *mm)
{
     /*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);
     list_init(&pra_list_head);
     curr_ptr = &pra_list_head;
     //cprintf("curr_ptr 0xffffffff%x\n", curr_ptr);
     mm->sm_priv = &pra_list_head;
     return 0;
}
/*
 * (3)_fifo_map_swappable: According FIFO PRA, we should link the most recent
arrival page at the back of pra_list_head qeueue
 */
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int
swap_in)
{
    //cprintf("swap_in:%d\n",swap_in);
    list_entry_t *entry=&(page->pra_page_link);

    assert(entry != NULL && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // 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_entry_t *head=(list_entry_t*) mm->sm_priv;
    //list_entry_t *entry=&(page->pra_page_link);
    //curr_ptr = entry;

    assert(entry != NULL && head != NULL);
    list_add_before(head, entry);
    page->visited = 1;
```

```
    return 0;
}
/*
 *  (4)_fifo_swap_out_victim: According FIFO PRA, we should unlink the  earliest
arrival page in front of pra_list_head qeueue,
 *                            then set the addr of addr of this page to ptr_page.
 */
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);
     /* 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: 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
        curr_ptr = list_next(curr_ptr);
        if(curr_ptr == head) {
            curr_ptr = list_next(curr_ptr);
            if(curr_ptr == head) {
                *ptr_page = NULL;
                break;
            }
        }
        struct Page* page = le2page(curr_ptr, pra_page_link);
        if(page->visited==0) {
            *ptr_page = page;
            list_del(curr_ptr);
            cprintf("curr_ptr %p\n",curr_ptr);
            break;
        }
        else {
            page->visited = 0;
        }
    }
    return 0;
}
static int
_clock_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
    //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);
    *(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
_clock_init(void)
{
    return 0;
}

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

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

struct swap_manager swap_manager_clock =
{
     .name            = "clock swap manager",
     .init            = &_clock_init,
     .init_mm         = &_clock_init_mm,
     .tick_event      = &_clock_tick_event,
     .map_swappable   = &_clock_map_swappable,
     .set_unswappable = &_clock_set_unswappable,
     .swap_out_victim = &_clock_swap_out_victim,
     .check_swap      = &_clock_check_swap,
};
```

设计和实现

Clock页面替换算法的设计和实现过程如下：

1. **初始化页面队列**：首先，在`_clock_init_mm`函数中，初始化一个页面队列`pra_list_head`，并初始化一个指向当前页面的指针`curr_ptr`，最初指向队列头。这个队列用于存储可替换的页面。

2. **页面访问标志**：每个页面结构（`struct Page`）中包含一个标志位`visited`，用于跟踪页面是否被访问。页面初始化时，`visited`被设置为0。

3. **页面访问时调用**：当页面被访问时，在`_clock_map_swappable`函数中，将页面插入到页面队列的末尾，并将页面的`visited`标志位设置为1，表示页面已被访问。

4. **页面替换时调用**：当需要替换页面时，在`_clock_swap_out_victim`函数中，使用Clock算法来选择替换的页面。Clock算法首先检查当前页面指针`curr_ptr`指向的页面是否已被访问。如果未被访问，则选择

这个页面进行替换，并将其从页面队列中删除。如果已被访问，则将其 `visited` 标志位重置为0，表示重新访问。

5. **页面替换完成**：选择页面后，将其指针赋给 `ptr_page` 以进行替换操作。

Clock和FIFO页替换算法区别

在Clock页替换算法和FIFO页替换算法之间存在一些关键区别，主要体现在页面选择的方式上。以下是它们之间的比较：

FIFO（First-In-First-Out）页面替换算法：

1. **页面选择方式**：FIFO算法按照页面进入内存的顺序来选择页面进行替换。它选择最早进入内存的页面进行替换，这是一个线性队列的方式。
2. **性能特点**：FIFO算法非常简单，容易实现，但在实际应用中性能较差。它容易受到Belady's Anomaly的影响，即在某些情况下，增加内存可能导致缺页率上升。
3. **不考虑页面访问频率**：FIFO不考虑页面的访问频率，而只考虑页面的进入顺序。

Clock页面替换算法：

1. **页面选择方式**：Clock算法使用一个环形队列（类似时钟的指针）来维护页面的访问情况。它根据页面的"访问位"来选择页面进行替换，而不是简单地按照进入内存的时间顺序。
2. **性能特点**：Clock算法相对于FIFO算法具有更好的性能。它能够更好地应对一些特定的工作负载，因为它考虑了页面的访问频率。
3. **考虑页面访问频率**：Clock算法通过维护"访问位"来跟踪页面的访问频率，如果一个页面被访问，它的"访问位"会被置为1。算法在选择替换页面时，首先查找访问位为0的页面，如果找不到，它会将所有页面的访问位都置为0，并选择第一个"访问位"为0的页面。

总的来说，Clock算法相对于FIFO算法来说，更复杂一些，但在性能上有一定的优势。它考虑了页面的访问频率，尽量保留常用的页面，以提高性能。然而，仍然存在一些特殊情况下，可能不如其他更高级的页面替换算法表现良好。选择页面替换算法通常要根据具体的应用和工作负载来进行权衡。

（五）练习 5：阅读代码和实现手册，理解页表映射方式相关知识

采用"一个大页"的页表映射方式与分级页表相比具有一些好处、优势，但也伴随着一些坏处和风险。以下是对这两种页表映射方式的比较：

好处和优势：

1. **减少页表数量**：使用"一个大页"可以减少页表的数量，因为一个大页可以映射更多的虚拟地址空间。这减少了页表的维护开销，包括内存占用和访问时间。
2. **提高访问效率**：减少页表数量可以提高访问效率，因为在寻找页表项时需要更少的级别访问。这降低了访问内存的开销，对于大型内存空间来说，性能提升明显。
3. **减少TLB缺失**："一个大页"可以减少翻译后备缓冲（TLB）的缺失率，因为更多的虚拟地址都映射到同一个大页中，减少了映射表项的变化，从而降低了TLB缺失率。

4. **适用于大内存应用**："一个大页"的页表映射方式特别适用于需要管理大量内存的应用，例如数据库管理系统和科学计算，因为它减少了页表的管理复杂性。

坏处和风险：

1. **内部碎片**：使用"一个大页"可能会导致内部碎片，因为即使应用程序只使用了一部分大页，整个大页也会被分配和映射，浪费了部分内存。
2. **不适用于小内存应用**：对于小内存应用，使用大页可能会导致浪费，因为大页的最小粒度较大，可能导致分配和映射更多内存。
3. **不适用于多任务系统**：在多任务系统中，不同任务可能需要不同的页表映射，使用"一个大页"可能会导致无法满足不同任务的需求。
4. **不适用于稀疏地址空间**："一个大页"的页表映射方式不适用于稀疏地址空间，因为它要求虚拟地址是连续的，否则将浪费内存。

综上所述，"一个大页"的页表映射方式在某些场景下具有明显的优势，可以提高性能和降低复杂性，特别适用于大内存应用。然而，它也有一些限制，不适用于小内存应用、多任务系统和稀疏地址空间。

（六）扩展练习 Challenge：实现不考虑实现开销和效率的 LRU 页替换算法

实现代码

```
#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, *curr_ptr;

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

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(head, entry);
    //visited为距离上一次访问的访问次数
    page->visited = 0;
    return 0;
}

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 *le = head->next;
    uint_t max = 0;
    // 遍历mm的链表，找出最后一个visited最大的值
    cprintf("\nPage swap out begin\n");
    while (le!=head) {
        struct Page* page = le2page(le,pra_page_link);
        if(page->visited >= max){
            max = page->visited;
            curr_ptr = le;
            struct Page* page2 = le2page(curr_ptr,pra_page_link);
        }
        cprintf("Page:%x,Page.visited:%d\n",page2ppn(page),page->visited);
        le = le->next;
    }
    *ptr_page = le2page(curr_ptr,pra_page_link);
    cprintf("vitim Page:%x,Page.visited:%d\n\n",page2ppn(*ptr_page),(*ptr_page)-
>visited);
    list_del(curr_ptr);
    return 0;
}


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

```
    cprintf("write Virt Page a in lru_check_swap\n");
    lru_write_memory(0x1000 ,0x0a);
    assert(pgfault_num==5);
    cprintf("write Virt Page b in lru_check_swap\n");
    lru_write_memory(0x2000 ,0x0b);
    assert(pgfault_num==5);
    cprintf("write Virt Page c in lru_check_swap\n");
    lru_write_memory(0x3000 ,0x0c);
    assert(pgfault_num==6);
    cprintf("write Virt Page d in lru_check_swap\n");
    lru_write_memory(0x4000 ,0x0d);
    assert(pgfault_num==7);
    cprintf("write Virt Page e in lru_check_swap\n");
    lru_write_memory(0x5000 ,0x0e);
    assert(pgfault_num==8);
    cprintf("write Virt Page a in lru_check_swap\n");
    assert(*(unsigned char *)0x1000 == 0x0a);
    lru_write_memory(0x1000 ,0x0a);
    assert(pgfault_num==9);
    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,
};
```

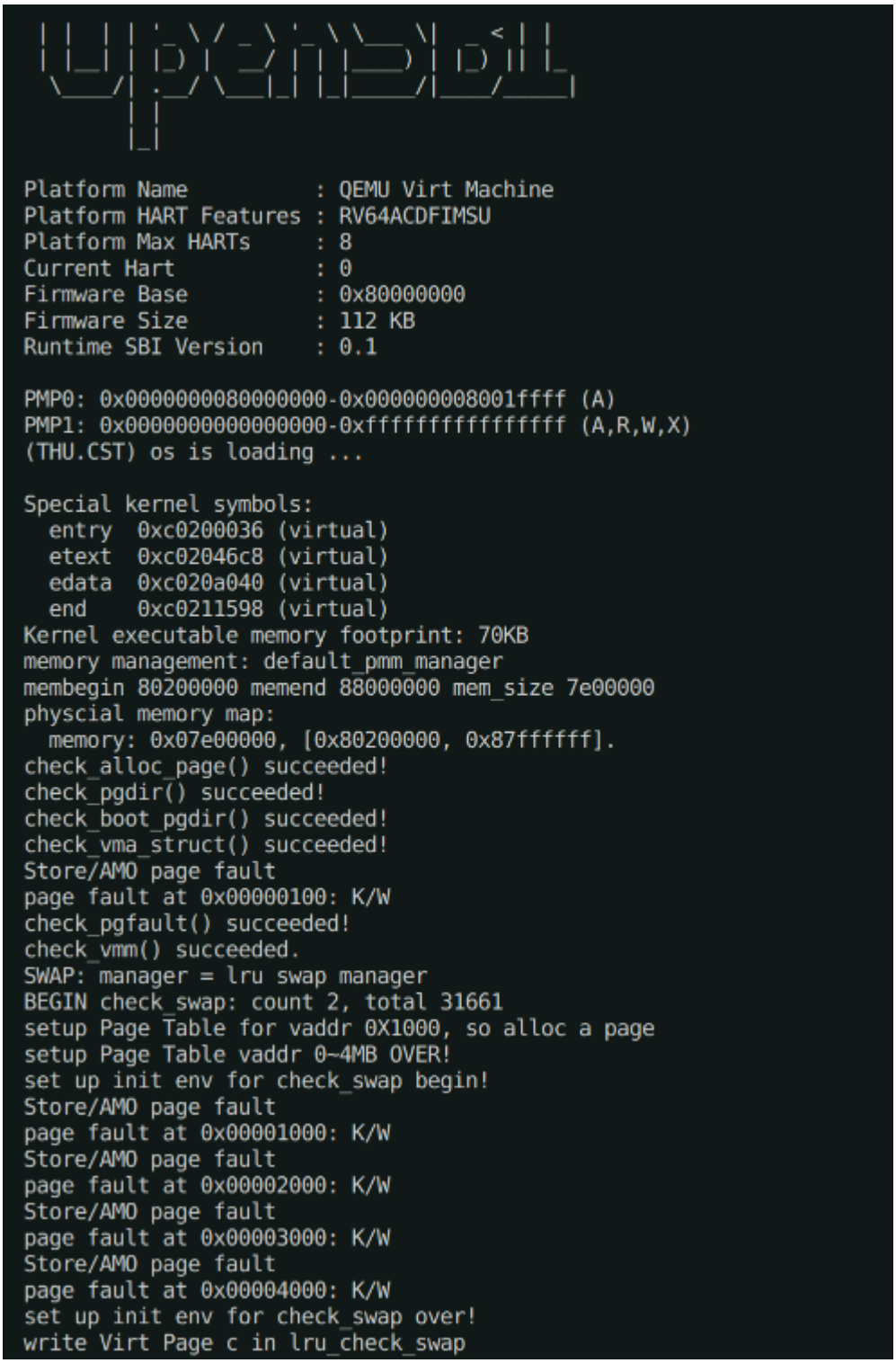
设计文档

以下是对我们实现的LRU页面置换算法的详细设计和分析：

1. 初始化（_lru_init_mm）：
    - `list_init(&pra_list_head);`：初始化了一个名为`pra_list_head`的链表。该链表用于维护页面的访问历史顺序，将最近访问的页面放在列表的末尾。
    - `mm->sm_priv = &pra_list_head;`：将初始化的链表与内存控制结构`mm`关联起来。允许算法从内存控制结构中访问LRU列表。
2. 映射可置换页面（_lru_map_swappable）：
    - `list_add(head, entry);`：将当前页面添加到LRU列表的末尾，标记它为最近使用的页面。这是LRU算法的核心操作，确保访问的页面被移到列表的末尾。
    - `page->visited = 0;`：将页面的访问计数初始化为零。访问计数可用于跟踪自上次访问以来的时间。
3. 替换受害者页面（_lru_swap_out_victim）：
    - `list_del(curr_ptr);`：删除LRU列表的前端页面，表示最久未使用的页面。该页面被选为替换的受害者。
    - 还维护一个`max`值，并通过迭代列表来查找具有最高`visited`计数的页面，确保选定的替换页面是最近最少使用的。
    - `*ptr_page = le2page(curr_ptr, pra_page_link);`：将`ptr_page`设置为将要替换出去的页面。
4. 检查置换（_lru_check_swap）：
    - 执行一系列内存写操作，模拟页面访问，以测试LRU算法。通过检查不同内存写入操作引起的页面缺页数（`pgfault_num`）来验证页面置换是否按预期工作。

经过分析，我们实现的算法选择替换页面时的时间复杂度为O(N)（其中N是系统中的页面数）。

测试结果



```
  | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
  | |__| | |_) |  __/ | | |____) | |_) || |_
   \____/| .__/ \___|_| |_|_____/|____/_____|
         | |
         |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc0200036 (virtual)
  etext  0xc02046c8 (virtual)
  edata  0xc020a040 (virtual)
  end    0xc0211598 (virtual)
Kernel executable memory footprint: 70KB
memory management: default_pmm_manager
membegin 80200000 memend 88000000 mem_size 7e00000
physcial memory map:
  memory: 0x07e00000, [0x80200000, 0x87ffffff].
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_vma_struct() succeeded!
Store/AMO page fault
page fault at 0x00000100: K/W
check_pgfault() succeeded!
check_vmm() succeeded.
SWAP: manager = lru swap manager
BEGIN check_swap: count 2, total 31661
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
Store/AMO page fault
page fault at 0x00001000: K/W
Store/AMO page fault
page fault at 0x00002000: K/W
Store/AMO page fault
page fault at 0x00003000: K/W
Store/AMO page fault
page fault at 0x00004000: K/W
set up init env for check_swap over!
write Virt Page c in lru_check_swap
```