Lab3:缺页异常和页面置换

做完实验二后，大家可以了解并掌握物理内存管理中页表的建立过程以及页面分配算法的具体实现。本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的**中断异常处理**机制，学习如何在磁盘上缓存内存页，从而能够支持虚拟内存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。

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。

实验内容

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成**Page Fault异常处理**和**部分页面替换算法**的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。如果大家有余力，可以尝试完成扩展练习，实现LRU页替换算法。

练习

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

练习0：填写已有实验

本实验依赖实验2。请把你做的实验2的代码填入本实验中代码中有“LAB2”的注释相应部分。（建议手动补充，不要直接使用merge）

> 貌似没找到要填写的地方？

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

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？

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？

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）。(提示:要输出curr_ptr的值才能通过make grade)

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。

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。

项目组成

表1：实验三文件列表

```
├── Makefile
├── kern
|   ├── debug
|   |   ├── assert.h
|   |   ├── kdebug.c
|   |   ├── kdebug.h
|   |   ├── kmonitor.c
```

```
|   |   ├── kmonitor.h
|   |   ├── panic.c
|   |   └── stab.h
|   ├── driver
|   |   ├── clock.c
|   |   ├── clock.h
|   |   ├── console.c
|   |   ├── console.h
|   |   ├── ide.c
|   |   ├── ide.h
|   |   ├── intr.c
|   |   └── intr.h
|   ├── fs
|   |   ├── fs.h
|   |   ├── swapfs.c
|   |   └── swapfs.h
|   ├── init
|   |   ├── entry.S
|   |   └── init.c
|   ├── libs
|   |   └── stdio.c
|   ├── mm
|   |   ├── default_pmm.c
|   |   ├── default_pmm.h
|   |   ├── memlayout.h
|   |   ├── mmu.h
|   |   ├── pmm.c
|   |   ├── pmm.h
|   |   ├── swap.c
|   |   ├── swap.h
|   |   ├── swap_clock.c
|   |   ├── swap_clock.h
|   |   ├── swap_fifo.c
|   |   ├── swap_fifo.h
|   |   ├── vmm.c
|   |   └── vmm.h
|   ├── sync
|   |   └── sync.h
|   └── trap
|       ├── trap.c
|       ├── trap.h
|       └── trapentry.S
├── lab3.md
├── libs
|   ├── atomic.h
|   ├── defs.h
|   ├── error.h
|   ├── list.h
|   ├── printfmt.c
|   ├── rand.c
|   ├── readline.c
|   ├── riscv.h
|   ├── sbi.h
|   ├── stdarg.h
|   ├── stdio.h
|   ├── stdlib.h
```

```
|   ├── string.c
|   └── string.h
└── tools
    ├── boot.ld
    ├── function.mk
    ├── gdbinit
    ├── grade.sh
    ├── kernel.ld
    ├── sign.c
    └── vector.c
```

编译方法

编译并运行代码的命令如下：

```
make

make qemu
```

则可以得到如下显示界面（仅供参考）

```
chenyu$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc0200036 (virtual)
  etext  0xc02042cc (virtual)
  edata  0xc020a040 (virtual)
  end    0xc02115a0 (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 = clock 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!
Store/AMO page fault
page fault at 0x00005000: K/W
curr_ptr 0xffffffffc02258a8
curr_ptr 0xffffffffc02258a8
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
Load page fault
page fault at 0x00001000: K/R
curr_ptr 0xffffffffc02258f0
curr_ptr 0xffffffffc02258f0
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
count is 1, total is 8
check_swap() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
……
```

通过上述运行结果，我们可以看到ucore在显示特殊内核符号地址后，展示了物理内存的映射信息，并进行了内存管理相关的检查。接着ucore进入页面置换和缺页异常处理的测试，并成功完成了页面置换测试。最后，ucore设置了定时器中断，在分页模式下响应时钟中断。

虚拟内存管理

基本原理概述

虚拟内存

什么是虚拟内存？简单地说是指程序员或CPU“看到”的内存。但有几点需要注意：

1. **虚拟内存单元不一定有实际的物理内存单元对应**，即实际的物理内存单元可能不存在；
2. 如果虚拟内存单元对应有实际的物理内存单元，那**二者的地址一般是不相等**的；
3. 通过操作系统实现的某种内存映射可建立虚拟内存与物理内存的对应关系，使得程序员或CPU访问的虚拟内存地址会自动转换为一个物理内存地址。

那么这个“虚拟”的作用或意义在哪里体现呢？在操作系统中，虚拟内存其实包含多个虚拟层次，在不同的层次体现了不同的作用。首先，在有了分页机制后，程序员或CPU“看到”的地址已经不是实际的物理地址了，这已经有一层虚拟化，我们可简称为**内存地址虚拟化**。有了内存地址虚拟化，我们就可以通过设置页表项来限定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为**按需分页**（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为**页换入换出**（page　swap in/out）。这种内存管理技术给了程序员更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

实验执行流程概述

本次实验主要完成ucore内核对虚拟内存的管理工作。首先我们要完成初始化虚拟内存管理机制，即需要设置好哪些页需要放在物理内存中，哪些页不需要放在物理内存中，而是可被换出到硬盘上，并涉及完善建立页表映射、页访问异常处理操作等函数的实现。然后执行一组访存测试，看看我们建立的页表项是否能够正确完成虚实地址映射，是否正确描述了虚拟内存页在物理内存中还是在硬盘上，是否能够正确把虚拟内存页在物理内存和硬盘之间进行传递，是否正确实现了页面替换算法等。lab3的总体执行流程如下：

首先，整个实验过程以ucore的总控函数init为起点。在初始化阶段，首先调用pmm_init函数完成物理内存的管理初始化。接下来，执行中断和异常相关的初始化工作。此过程涉及调用pic_init函数和idt_init函数，用于初始化处理器中断控制器（PIC）和中断描述符表（IDT），与之前的lab1中断和异常初始化工作相同。 随后，调用vmm_init函数进行虚拟内存管理机制的初始化。在此阶段，主要是建立虚拟地址到物理地址的映射关系，为虚拟内存提供管理支持。继续执行初始化过程，接下来调用ide_init函数完成对用于页面换入和换出的硬盘（通常称为swap硬盘）的初始化工作。在这个阶段，ucore准备好了对硬盘数据块的读写操作，以便后续页面置换算法的实现。 最后，完成整个初始化过程，调用swap_init函数用于初始化页面置换算法，这其中包括Clock页替换算法的相关数据结构和初始化步骤。通过swap_init，ucore确保页面置换算法准备就绪，可以在需要时执行页面换入和换出操作，以优化内存的利用。

下面我们就来看看如何使用多级页表进行虚拟内存管理和页面置换，ucore在实现上述技术时，需要解决三个关键问题：

1. 当程序运行中访问内存产生page fault异常时，如何判定这个引起异常的虚拟地址内存访问是越界、写只读页的“非法地址”访问还是由于数据被临时换出到磁盘上或还没有分配内存的“合法地址”访问?
2. 何时进行请求调页/页换入换出处理?
3. 如何在现有ucore的基础上实现页替换算法?

接下来将进一步分析完成lab3主要注意的关键问题和涉及的关键数据结构。

关键数据结构和相关函数分析

大家在之前的实验中都尝试了`make qemu`这条指令，但实际上我们在QEMU里并没有真正模拟“硬盘”。为了实现“页面置换”的效果，我们采取的措施是，从内核的静态存储(static)区里面分出一块内存， 声称这块存储区域是”硬盘“，然后包裹一下给出”硬盘IO“的接口。 这听上去很令人费解，但如果仔细思考一下，内存和硬盘，除了一个掉电后数据易失一个不易失，一个访问快一个访问慢，其实并没有本质的区别。对于我们的页面置换算法来说，也不要求硬盘上存多余页面的交换空间能够“不易失”，反正这些页面存在内存里的时候就是易失的。理论上，我们完全可以把一块机械硬盘加以改造，写好驱动之后，插到主板的内存插槽上作为内存条使用，当然我们要忽视性能方面的差距。那么我们就把QEMU模拟出来的一块ram叫做“硬盘”，用作页面置换时的交换区，完全没有问题。你可能会觉得，这样折腾一通，我们总共能使用的页面数并没有增加，原先能直接在内存里使用的一些页面变成了“硬盘”，只是在自娱自乐。这样的想法当然是没错的，不过我们在这里只是想介绍页面置换的原理，而并不关心实际性能。

那么模拟二等这个过程我们在`driver/ide.h` `driver/ide.c` `fs/fs.h` `fs/swapfs.h` `fs/swapfs.c`通过修改代码一步步实现。

首先我们先了解一下各个文件名的含义，虽然这些文件名你可以自定义设置，但我们还是一般采用比较规范的方式进行命名,这样可以帮助你更好的理解项目结构。

`ide`在这里不是integrated development environment的意思，而是Integrated Drive Electronics的意思，表示的是一种标准的硬盘接口。我们这里写的东西和Integrated Drive Electronics并不相关，这个命名是ucore的历史遗留。

`fs`全称为file system,我们这里其实并没有“文件”的概念，这个模块称作`fs`只是说明它是“硬盘”和内核之间的接口。

```
// kern/driver/ide.c
/*
#include"s
*/

void ide_init(void) {}

#define MAX_IDE 2
#define MAX_DISK_NSECS 56
static char ide[MAX_DISK_NSECS * SECTSIZE];

bool ide_device_valid(unsigned short ideno) { return ideno < MAX_IDE; }

size_t ide_device_size(unsigned short ideno) { return MAX_DISK_NSECS; }

int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
                  size_t nsecs) {
    //ideno: 假设挂载了多块磁盘，选择哪一块磁盘 这里我们其实只有一块“磁盘”，这个参数就没用到
    int iobase = secno * SECTSIZE;
    memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
    return 0;
}

int ide_write_secs(unsigned short ideno, uint32_t secno, const void *src,
                   size_t nsecs) {
    int iobase = secno * SECTSIZE;
    memcpy(&ide[iobase], src, nsecs * SECTSIZE);
    return 0;
}
```

可以看到，我们这里所谓的“硬盘IO”，只是在内存里用`memcpy`把数据复制来复制去。同时为了逼真地模仿磁盘，我们只允许以磁盘扇区为数据传输的基本单位，也就是一次传输的数据必须是512字节的倍数，并且必须对齐。

```
// kern/fs/fs.h
#ifndef __KERN_FS_FS_H__
#define __KERN_FS_FS_H__

#include <mmu.h>

#define SECTSIZE            512
#define PAGE_NSECT          (PGSIZE / SECTSIZE) //一页需要几个磁盘扇区?

#define SWAP_DEV_NO         1

#endif /* !__KERN_FS_FS_H__ */

// kern/mm/swap.h
extern size_t max_swap_offset;
// kern/mm/swap.c
size_t max_swap_offset;
```

```
// kern/fs/swapfs.c
#include <swap.h>
#include <swapfs.h>
#include <mmu.h>
#include <fs.h>
#include <ide.h>
#include <pmm.h>
#include <assert.h>

void swapfs_init(void) { //做一些检查
    static_assert((PGSIZE % SECTSIZE) == 0);
    if (!ide_device_valid(SWAP_DEV_NO)) {
        panic("swap fs isn't available.\n");
    }
    //swap.c/swap.h里的全局变量
    max_swap_offset = ide_device_size(SWAP_DEV_NO) / (PAGE_NSECT);
}

int swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}

int swapfs_write(swap_entry_t entry, struct Page *page) {
    return ide_write_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

页表项设计思路

我们定义一些对sv39页表项（Page Table Entry）进行操作的宏。这里的定义和我们在Lab2里介绍的定义一致。

有时候我们把多级页表中较高级别的页表（“页表的页表”）叫做`Page Directory`。在实验二中我们知道sv39中采用的是三级页表，那么在这里我们把页表项里从高到低三级页表的页码分别称作PDX1, PDX0和PTX(Page Table Index)。

```
// kern/mm/mmu.h
#ifndef __KERN_MM_MMU_H__
#define __KERN_MM_MMU_H__

#ifndef __ASSEMBLER__
#include <defs.h>
#endif /* !__ASSEMBLER__ */

// A linear address 'la' has a four-part structure as follows:
//
// +--------9-------+-------9--------+-------9--------+---------12----------+
// | Page Directory | Page Directory |   Page Table   | Offset within Page  |
// |     Index 1    |    Index 2     |                |                     |
// +----------------+----------------+----------------+---------------------+
//  \-- PDX1(la) --/ \-- PDX0(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \------------------PPN(la)----------------------/
//
```

```
// The PDX1, PDX0, PTX, PGOFF, and PPN macros decompose linear addresses as
shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).

// RISC-V uses 39-bit virtual address to access 56-bit physical address!
// Sv39 virtual address:
// +----9----+----9---+----9---+---12--+
// |  VPN[2] | VPN[1] | VPN[0] | PGOFF |
// +---------+----+---+--------+-------+
//
// Sv39 physical address:
// +----26---+----9---+----9---+---12--+
// |  PPN[2] | PPN[1] | PPN[0] | PGOFF |
// +---------+----+---+--------+-------+
//
// Sv39 page table entry:
// +----26---+----9---+----9---+---2----+-------8-------+
// |  PPN[2] | PPN[1] | PPN[0] |Reserved|D|A|G|U|X|W|R|V|
// +---------+----+---+--------+--------+---------------+

// page directory index
#define PDX1(la) ((((uintptr_t)(la)) >> PDX1SHIFT) & 0x1FF)
#define PDX0(la) ((((uintptr_t)(la)) >> PDX0SHIFT) & 0x1FF)

// page table index
#define PTX(la) ((((uintptr_t)(la)) >> PTXSHIFT) & 0x1FF)

// page number field of address
#define PPN(la) (((uintptr_t)(la)) >> PTXSHIFT)

// offset in page
#define PGOFF(la) (((uintptr_t)(la)) & 0xFFF)

// construct linear address from indexes and offset
#define PGADDR(d1, d0, t, o) ((uintptr_t)((d1) << PDX1SHIFT | (d0) << PDX0SHIFT |
(t) << PTXSHIFT | (o)))

// address in page table or page directory entry
// 把页表项里存储的地址拿出来
#define PTE_ADDR(pte)   (((uintptr_t)(pte) & ~0x3FF) << (PTXSHIFT -
PTE_PPN_SHIFT))
#define PDE_ADDR(pde)   PTE_ADDR(pde)

/* page directory and page table constants */
#define NPDEENTRY       512                     // page directory entries per page
directory
#define NPTEENTRY       512                     // page table entries per page
table

#define PGSIZE          4096                    // bytes mapped by a page
#define PGSHIFT         12                      // log2(PGSIZE)
#define PTSIZE          (PGSIZE * NPTEENTRY)    // bytes mapped by a page
directory entry
#define PTSHIFT         21                      // log2(PTSIZE)
```

```
#define PTXSHIFT        12                      // offset of PTX in a linear
address
#define PDX0SHIFT       21                      // offset of PDX0 in a linear
address
#define PDX1SHIFT       30                      // offset of PDX0 in a linear
address
#define PTE_PPN_SHIFT   10                      // offset of PPN in a physical
address

// page table entry (PTE) fields
#define PTE_V     0x001 // Valid
#define PTE_R     0x002 // Read
#define PTE_W     0x004 // Write
#define PTE_X     0x008 // Execute
#define PTE_U     0x010 // User
```

上述代码定义了一些与内存管理单元（Memory Management Unit, MMU）相关的宏和常量，用于操作线性地址和物理地址，以及页表项的字段。最后我们看一下`kern/init/init.c`里的变化:

```
// kern/init/init.c
int kern_init(void){
    /* blabla */
    pmm_init();                 // init physical memory management
                                // 我们加入了多级页表的接口和测试

    idt_init();                 // init interrupt descriptor table

    vmm_init();                 // init virtual memory management
                                // 新增函数，初始化虚拟内存管理并测试

    ide_init();                 // init ide devices. 新增函数，初始化"硬盘".
                                //其实这个函数啥也没做，属于"历史遗留"

    swap_init();                // init swap. 新增函数，初始化页面置换机制并测试

    clock_init();               // init clock interrupt
    /* blabla */
}
```

可以看到的是我们在原本的基础上又新增了一个用来初始化“硬盘”的函数`ide_init()`。

使用多级页表实现虚拟存储

要想实现虚拟存储，我们需要把页表放在内存里，并且需要有办法修改页表，比如在页表里增加一个页面的映射或者删除某个页面的映射。

要想实现页面映射，我们最主要需要修改的是两个接口：

- `page_insert()`，在页表里建立一个映射
- `page_remove()`，在页表里删除一个映射

这些内容都需要在`kern/mm/pmm.c`里面编写。然后我们可以在虚拟内存空间的第一个大大页(Giga Page)中建立一些映射来做测试。通过编写`page_ref()`函数用来检查映射关系是否实现，这个函数会返回一个物理页面被多少个虚拟页面所对应。

```
static void check_pgdir(void) {
    // assert(npage <= KMEMSIZE / PGSIZE);
    // The memory starts at 2GB in RISC-V
    // so npage is always larger than KMEMSIZE / PGSIZE
    assert(npage <= KERNTOP / PGSIZE);
    //boot_pgdir是页表的虚拟地址
    assert(boot_pgdir != NULL && (uint32_t)PGOFF(boot_pgdir) == 0);
    assert(get_page(boot_pgdir, 0x0, NULL) == NULL);
    //get_page()尝试找到虚拟内存0x0对应的页，现在当然是没有的，返回NULL

    struct Page *p1, *p2;
    p1 = alloc_page();//拿过来一个物理页面
    assert(page_insert(boot_pgdir, p1, 0x0, 0) == 0);//把这个物理页面通过多级页表映射
到0x0
    pte_t *ptep;
    assert((ptep = get_pte(boot_pgdir, 0x0, 0)) != NULL);
    assert(pte2page(*ptep) == p1);
    assert(page_ref(p1) == 1);

    ptep = (pte_t *)KADDR(PDE_ADDR(boot_pgdir[0]));
    ptep = (pte_t *)KADDR(PDE_ADDR(ptep[0])) + 1;
    assert(get_pte(boot_pgdir, PGSIZE, 0) == ptep);
    //get_pte查找某个虚拟地址对应的页表项，如果不存在这个页表项，会为它分配各级的页表

    p2 = alloc_page();
    assert(page_insert(boot_pgdir, p2, PGSIZE, PTE_U | PTE_W) == 0);
    assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
    assert(*ptep & PTE_U);
    assert(*ptep & PTE_W);
    assert(boot_pgdir[0] & PTE_U);
    assert(page_ref(p2) == 1);

    assert(page_insert(boot_pgdir, p1, PGSIZE, 0) == 0);
    assert(page_ref(p1) == 2);
    assert(page_ref(p2) == 0);
    assert((ptep = get_pte(boot_pgdir, PGSIZE, 0)) != NULL);
    assert(pte2page(*ptep) == p1);
    assert((*ptep & PTE_U) == 0);

    page_remove(boot_pgdir, 0x0);
    assert(page_ref(p1) == 1);
    assert(page_ref(p2) == 0);

    page_remove(boot_pgdir, PGSIZE);
    assert(page_ref(p1) == 0);
    assert(page_ref(p2) == 0);

    assert(page_ref(pde2page(boot_pgdir[0])) == 1);
    free_page(pde2page(boot_pgdir[0]));
    boot_pgdir[0] = 0;//清除测试的痕迹

    cprintf("check_pgdir() succeeded!\n");
}
```

在映射关系建立完成后，如何新增一个映射关系和删除一个映射关系也是非常重要的内容，我们来看`page_insert()`,`page_remove()`的实现。它们都涉及到调用两个对页表项进行操作的函数：`get_pte()`和`page_remove_pte()`

```
int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    //pgdir是页表基址(satp)，page对应物理页面，la是虚拟地址
    pte_t *ptep = get_pte(pgdir, la, 1);
    //先找到对应页表项的位置，如果原先不存在，get_pte()会分配页表项的内存
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page);//指向这个物理页面的虚拟地址增加了一个
    if (*ptep & PTE_V) { //原先存在映射
        struct Page *p = pte2page(*ptep);
        if (p == page) {//如果这个映射原先就有
            page_ref_dec(page);
        } else {//如果原先这个虚拟地址映射到其他物理页面，那么需要删除映射
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = pte_create(page2ppn(page), PTE_V | perm);//构造页表项
    tlb_invalidate(pgdir, la);//页表改变之后要刷新TLB
    return 0;
}
void page_remove(pde_t *pgdir, uintptr_t la) {
    pte_t *ptep = get_pte(pgdir, la, 0);//找到页表项所在位置
    if (ptep != NULL) {
        page_remove_pte(pgdir, la, ptep);//删除这个页表项的映射
    }
}
//删除一个页表项以及它的映射
static inline void page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if (*ptep & PTE_V) {  //(1) check if this page table entry is valid
        struct Page *page = pte2page(*ptep);  //(2) find corresponding page to
pte
        page_ref_dec(page);   //(3) decrease page reference
        if (page_ref(page) == 0) {
            //(4) and free this page when page reference reachs 0
            free_page(page);
        }
        *ptep = 0;                  //(5) clear page table entry
        tlb_invalidate(pgdir, la);  //(6) flush tlb
    }
}
//寻找(有必要的时候分配)一个页表项
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /* LAB2 EXERCISE 2: YOUR CODE
     *
     * If you need to visit a physical address, please use KADDR()
     * please read pmm.h for useful macros
     *
     * Maybe you want help comment, BELOW comments can help you finish the code
     *
     * Some Useful MACROs and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
```

```
     *   PDX(la) = the index of page directory entry of VIRTUAL ADDRESS la.
     *   KADDR(pa) : takes a physical address and returns the corresponding
     * kernel virtual address.
     *   set_page_ref(page,1) : means the page be referenced by one time
     *   page2pa(page): get the physical address of memory which this (struct
     * Page *) page  manages
     *   struct Page * alloc_page() : allocation a page
     *   memset(void *s, char c, size_t n) : sets the first n bytes of the
     * memory area pointed by s
     *                                       to the specified value c.
     * DEFINEs:
     *   PTE_P           0x001                   // page table/directory entry
     * flags bit : Present
     *   PTE_W           0x002                   // page table/directory entry
     * flags bit : Writeable
     *   PTE_U           0x004                   // page table/directory entry
     * flags bit : User can access
     */
    pde_t *pdep1 = &pgdir[PDX1(la)];//找到对应的Giga Page
    if (!(*pdep1 & PTE_V)) {//如果下一级页表不存在，那就给它分配一页，创造新页表
        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);
        //我们现在在虚拟地址空间中，所以要转化为KADDR再memset.
        //不管页表怎么构造，我们确保物理地址和虚拟地址的偏移量始终相同，那么就可以用这种方式完成
对物理内存的访问。
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//注意这里R,W,X全零
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//再下一级页表
    //这里的逻辑和前面完全一致，页表不存在就现在分配一个
    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);
    }
    //找到输入的虚拟地址la对应的页表项的地址(可能是刚刚分配的)
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

在`entry.S`里，我们虽然构造了一个简单映射使得内核能够运行在虚拟空间上，但是这个映射是比较粗糙的。

我们知道一个程序通常含有下面几段：

- `.text`段：存放代码，需要是可读、可执行的，但不可写。
- `.rodata`段：存放只读数据，顾名思义，需要可读，但不可写亦不可执行。

- `.data` 段：存放经过初始化的数据，需要可读、可写。
- `.bss` 段：存放经过零初始化的数据，需要可读、可写。与 `.data` 段的区别在于由于我们知道它被零初始化，因此在可执行文件中可以只存放该段的开头地址和大小而不用存全为0的数据。在执行时由操作系统进行处理。

我们看到各个段需要的访问权限是不同的。但是现在使用一个大大页(Giga Page)进行映射时，它们都拥有相同的权限，那么在现在的映射下，我们甚至可以修改内核 `.text` 段的代码，因为我们通过一个标志位 `W=1` 的页表项就可以完成映射，但这显然会带来安全隐患。

因此，我们考虑对这些段分别进行重映射，使得他们的访问权限可以被正确设置。虽然还是每个段都还是映射以同样的偏移量映射到相同的地方，但实现过程需要更加精细。

这里还有一个小坑：对于我们最开始已经用特殊方式映射的一个大大页(Giga Page)，该怎么对那里面的地址重新进行映射？这个过程比较麻烦。但大家可以基本理解为放弃现有的页表，直接新建一个页表，在新页表里面完成重映射，然后把satp指向新的页表，这样就实现了重新映射。

处理缺页异常（page fault异常）

什么是缺页异常?

缺页异常是指CPU访问的虚拟地址时， MMU没有办法找到对应的物理地址映射关系，或者与该物理页的访问权不一致而发生的异常。

CPU通过地址总线可以访问连接在地址总线上的所有外设，包括物理内存、IO设备等等，但从CPU发出的访问地址并非是这些外设在地址总线上的物理地址，而是一个虚拟地址，由MMU将虚拟地址转换成物理地址再从地址总线上发出，MMU上的这种虚拟地址和物理地址的转换关系是需要创建的，并且还需要设置这个物理页的访问权限。

在实验二中我们有关内存的所有数据结构和相关操作都是直接针对实际存在的资源，即针对物理内存空间的管理，而没有从一般应用程序对内存的“需求”考虑，所以我们需要有相关的数据结构和操作来体现一般应用程序对虚拟内存的“需求”。而一般应用程序的对虚拟内存的“需求”与物理内存空间的“供给”没有直接的对应关系，在ucore中我们是通过page fault异常处理来间接完成这二者之间的衔接。但需要注意的是在lab3中仅实现了简单的页面置换机制，还没有涉及lab4和lab5才实现的内核线程和用户进程，所以还无法通过内核线程机制实现一个完整意义上的虚拟内存页面置换功能。

> **须知：哪些页面可以被换出?**
>
> 在操作系统的设计中，一个基本的原则是：并非所有的物理页都可以交换出去的，只有映射到用户空间且被用户程序直接访问的页面才能被交换，而被内核直接使用的内核空间的页面不能被换出。这里面的原因是什么呢？操作系统是执行的关键代码，需要保证运行的高效性和实时性，如果在操作系统执行过程中，发生了缺页现象，则操作系统不得不等很长时间（硬盘的访问速度比内存的访问速度慢 2~3 个数量级），这将导致整个系统运行低效。而且，不难想象，处理缺页过程所用到的内核代码或者数据如果被换出，整个内核都面临崩溃的危险。
>
> 但在实验三实现的 ucore 中，我们只是实现了换入换出机制，还没有设计用户态执行的程序，所以我们在实验三中仅仅通过执行 check_swap 函数在内核中分配一些页，模拟对这些页的访问，然后通过 do_pgfault 来调用 swap_map_swappable 函数来查询这些页的访问情况并间接调用相关函数，换出“不常用”的页到磁盘上。

当我们引入了虚拟内存，就意味着虚拟内存的空间可以远远大于物理内存，也意味着程序可以访问"不对应物理内存页帧的虚拟内存地址"，这时CPU应当抛出 `Page Fault` 这个异常。

回想一下，我们处理异常的时候，是在`kern/trap/trap.c`的`exception_handler()`函数里进行的。按照`scause`寄存器对异常的分类里，有`CAUSE_LOAD_PAGE_FAULT`和`CAUSE_STORE_PAGE_FAULT`两个case。之前我们并没有真正对异常进行处理，只是简单输出一下就返回了，但现在我们要真正进行Page Fault的处理。

```
// kern/trap/trap.c
static inline void print_pgfault(struct trapframe *tf) {
    cprintf("page fault at 0x%08x: %c/%c\n", tf->badvaddr,
            trap_in_kernel(tf) ? 'K' : 'U',
            tf->cause == CAUSE_STORE_PAGE_FAULT ? 'W' : 'R');
}

static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}

void exception_handler(struct trapframe *tf) {
    int ret;
    switch (tf->cause) {
        /* .... other cases */
        case CAUSE_FETCH_PAGE_FAULT:// 取指令时发生的Page Fault先不处理
            cprintf("Instruction page fault\n");
            break;
        case CAUSE_LOAD_PAGE_FAULT:
            cprintf("Load page fault\n");
            if ((ret = pgfault_handler(tf)) != 0) {
                print_trapframe(tf);
                panic("handle pgfault failed. %e\n", ret);
            }
            break;
        case CAUSE_STORE_PAGE_FAULT:
            cprintf("Store/AMO page fault\n");
            if ((ret = pgfault_handler(tf)) != 0) { //do_pgfault()页面置换成功时返回
0
                print_trapframe(tf);
                panic("handle pgfault failed. %e\n", ret);
            }
            break;
        default:
            print_trapframe(tf);
            break;
    }
}
```

这里的异常处理程序会把Page Fault分发给`kern/mm/vmm.c`的`do_pgfault()`函数并尝试进行页面置换。

之前我们进行物理页帧管理时有个功能没有实现，那就是动态的内存分配。管理虚拟内存的数据结构（页表）需要有空间进行存储，而我们又没有给它预先分配内存（也无法预先分配，因为事先不确定我们的页表需要分配多少内存），于是我们就需要设置接口来负责分配释放内存，这里我们选择的是`malloc/free`的接口。同样，我们也要在`pmm.h`里编写对物理页面和虚拟地址，物理地址进行转换的一些函数。

```
// kern/mm/pmm.c
void *kmalloc(size_t n) { //分配至少n个连续的字节，这里实现得不精细，占用的只能是整数个页。
    void *ptr = NULL;
    struct Page *base = NULL;
    assert(n > 0 && n < 1024 * 0124);
    int num_pages = (n + PGSIZE - 1) / PGSIZE; //向上取整到整数个页
    base = alloc_pages(num_pages);
    assert(base != NULL); //如果分配失败就直接panic
    ptr = page2kva(base); //分配的内存的起始位置（虚拟地址），
    //page2kva，就是page_to_kernel_virtual_address
    return ptr;
}

void kfree(void *ptr, size_t n) { //从某个位置开始释放n个字节
    assert(n > 0 && n < 1024 * 0124);
    assert(ptr != NULL);
    struct Page *base = NULL;
    int num_pages = (n + PGSIZE - 1) / PGSIZE;
    /*计算num_pages和kmalloc里一样，
    但是如果程序员写错了呢？调用kfree的时候传入的n和调用kmalloc传入的n不一样？
    就像你平时在windows/linux写C语言一样，会出各种奇奇怪怪的bug。
    */
    base = kva2page(ptr);//kernel_virtual_address_to_page
    free_pages(base, num_pages);
}
// kern/mm/pmm.h
/*
KADDR，PADDR进行的是物理地址和虚拟地址的互换
由于我们在ucore里实现的页表映射很简单，所有物理地址和虚拟地址的偏移值相同，
所以这两个宏本质上只是做了一步加法/减法，额外还做了一些合法性检查。
*/
/* *
 * PADDR - takes a kernel virtual address (an address that points above
 * KERNBASE),
 * where the machine's maximum 256MB of physical memory is mapped and returns
 * the
 * corresponding physical address.  It panics if you pass it a non-kernel
 * virtual address.
 * */
#define PADDR(kva)                                                  \
    ({                                                              \
        uintptr_t __m_kva = (uintptr_t)(kva);                       \
        if (__m_kva < KERNBASE) {                                   \
            panic("PADDR called with invalid kva %08lx", __m_kva); \
        }                                                           \
        __m_kva - va_pa_offset;                                     \
    })

/* *
```

```
 * KADDR - takes a physical address and returns the corresponding kernel virtual
 * address. It panics if you pass an invalid physical address.
 * */
#define KADDR(pa)                                                   \
    ({                                                              \
        uintptr_t __m_pa = (pa);                                    \
        size_t __m_ppn = PPN(__m_pa);                               \
        if (__m_ppn >= npage) {                                     \
            panic("KADDR called with invalid pa %08lx", __m_pa);    \
        }                                                           \
        (void *)(__m_pa + va_pa_offset);                            \
    })

extern struct Page *pages;
extern size_t npage;
extern const size_t nbase;
extern uint_t va_pa_offset;

/*
我们曾经在内存里分配了一堆连续的Page结构体，来管理物理页面。可以把它们看作一个结构体数组。
pages指针是这个数组的起始地址，减一下，加上一个基准值nbase，就可以得到正确的物理页号。
pages指针和nbase基准值我们都在其他地方做了正确的初始化
*/
static inline ppn_t page2ppn(struct Page *page) { return page - pages + nbase; }
/*
指向某个Page结构体的指针，对应一个物理页面，也对应一个起始的物理地址。
左移若干位就可以从物理页号得到页面的起始物理地址。
*/
static inline uintptr_t page2pa(struct Page *page) {
    return page2ppn(page) << PGSHIFT;
}
/*
倒过来，从物理页面的地址得到所在的物理页面。实际上是得到管理这个物理页面的Page结构体。
*/
static inline struct Page *pa2page(uintptr_t pa) {
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa");
    }
    return &pages[PPN(pa) - nbase];//把pages指针当作数组使用
}

static inline void *page2kva(struct Page *page) { return KADDR(page2pa(page)); }

static inline struct Page *kva2page(void *kva) { return pa2page(PADDR(kva)); }

//从页表项得到对应的页，这里用到了 PTE_ADDR(pte)宏，对页表项做操作，在mmu.h里      定义
static inline struct Page *pte2page(pte_t pte) {
    if (!(pte & PTE_V)) {
        panic("pte2page called with invalid pte");
    }
    return pa2page(PTE_ADDR(pte));
}
//PDE(Page Directory Entry)指的是不在叶节点的页表项（指向低一级页表的页表项）
static inline struct Page *pde2page(pde_t pde) { //PDE_ADDR这个宏和PTE_ADDR是一样的
    return pa2page(PDE_ADDR(pde));
}
```

接下来需要我们处理的是多级页表。之前的初始页表占据一个页的物理内存，只有一个页表项是有用的，映射了一个大大页(Giga Page)。我们将在接下来的内容中实现页面置换机制。

页面置换机制的实现

页替换算法设计思路

操作系统为何要进行页面置换呢？这是由于操作系统给用户态的应用程序提供了一个虚拟的“大容量”内存空间，而实际的物理内存空间又没有那么大。所以操作系统就就“瞒着”应用程序，只把应用程序中“常用”的数据和代码放在物理内存中，而不常用的数据和代码放在了硬盘这样的存储介质上。如果应用程序访问的是“常用”的数据和代码，那么操作系统已经放置在内存中了，不会出现什么问题。但当应用程序访问它认为应该在内存中的的数据或代码时，如果这些数据或代码不在内存中，则根据上一小节的介绍，会产生页访问异常。这时，操作系统必须能够应对这种页访问异常，即尽快把应用程序当前需要的数据或代码放到内存中来，然后重新执行应用程序产生异常的访存指令。如果在把硬盘中对应的数据或代码调入内存前，操作系统发现物理内存已经没有空闲空间了，这时操作系统必须把它认为“不常用”的页换出到磁盘上去，以腾出内存空闲空间给应用程序所需的数据或代码。

操作系统迟早会碰到没有内存空闲空间而必须要置换出内存中某个“不常用”的页的情况。如何判断内存中哪些是“常用”的页，哪些是“不常用”的页，把“常用”的页保持在内存中，在物理内存空闲空间不够的情况下，把“不常用”的页置换到硬盘上就是页替换算法着重考虑的问题。容易理解，一个好的页替换算法会导致页访问异常次数少，也就意味着访问硬盘的次数也少，从而使得应用程序执行的效率就高。

本次实验涉及的页替换算法（包括扩展练习）：

- 先进先出(First In First Out, FIFO)页替换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO 算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO 算法的另一个缺点是，它有一种异常现象（Belady 现象），即在增加放置页的物理页帧的情况下，反而使页访问异常次数增多。
- 最久未使用(least recently used, LRU)算法：利用局部性，通过过去的访问情况预测未来的访问情况，我们可以认为最近还被访问过的页面将来被访问的可能性大，而很久没访问过的页面将来不太可能被访问。于是我们比较当前内存里的页面最近一次被访问的时间，把上一次访问时间离现在最久的页面置换出去。
- **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。**
- **改进的时钟（Enhanced Clock）页替换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面，所以优先淘汰没有修改的页，减少磁盘操作次数。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU 中**

的 MMU 硬件将把访问位置“1”。当该页被“写”时，CPU 中的 MMU 硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的 I/O 操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。

页面置换机制设计思路

现在我们来看看ucore页面置换机制的实现。

页面置换机制中， 我们需要维护一些”不在内存当中但是也许会用到“的页，它们存储在磁盘的交换区里，也有对应的虚拟地址，但是因为它们不在内存里，在页表里并没有对它们的虚拟地址进行映射。但是在发生Page Fault之后，会把访问到的页放到内存里，这时也许会把其他页扔出去，来给要用的页腾出地方。页面置换算法的核心任务，主要就是确定”把谁扔出去“。

页表里的信息大家都知道内容方面是比较有限的，基本上可以理解为"当前哪些数据在内存条里以及它们物理地址和虚拟地址的对应关系", 这里我们显然需要一些页表之外的数据结构来维护当前页表里没映射的页。也就是要存储以下这些信息：

- 分别在磁盘上的哪个位置?
- 有哪些虚拟地址对应的页面当前放在内存里?

这两类页面（位于内存/磁盘）因为会相互转换(换入/换出内存)，所以我们将这两类页面一起维护，也就是维护”所有可用的虚拟地址/虚拟页的集合“（不论当前这个虚拟地址对应的页在内存上还是在硬盘上）。之后我们将要实现进程机制，对于不同的进程，可用的虚拟地址（虚拟页）的集合常常是不一样的，因此每个进程需要一个页表，也需要一个数据结构来维护“所有可用的虚拟地址”。

因此，我们在vmm.h定义两个结构体 (vmm：virtural memory management)。

- `vma_struct` 结构体描述一段连续的虚拟地址，从 `vm_start` 到 `vm_end` 。 通过包含一个 `list_entry_t` 成员，我们可以把同一个页表对应的多个 `vma_struct` 结构体串成一个链表，在链表里把它们按照区间的起始点进行排序。
- `vm_flags` 表示的是一段虚拟地址对应的权限（可读，可写，可执行等），这个权限在页表项里也要进行对应的设置。

我们注意到，每个页表（每个虚拟地址空间）可能包含多个 `vma_struct`, 也就是多个访问权限可能不同的，不相交的连续地址区间。我们用 `mm_struct` 结构体把一个页表对应的信息组合起来，包括 `vma_struct` 链表的首指针，对应的页表在内存里的指针， `vma_struct` 链表的元素个数。

（参考 `libs/list.h` ）

```
// kern/mm/vmm.h
//pre define
struct mm_struct;

// the virtual continuous memory area(vma), [vm_start, vm_end),
// addr belong to a vma means  vma.vm_start<= addr <vma.vm_end
struct vma_struct {
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    uintptr_t vm_start;      // start addr of vma
    uintptr_t vm_end;        // end addr of vma, not include the vm_end itself
    uint_t vm_flags;       // flags of vma
    list_entry_t list_link;  // linear list link which sorted by start addr of
vma
};
```

```
#define le2vma(le, member)                  \
    to_struct((le), struct vma_struct, member)

#define VM_READ                 0x00000001
#define VM_WRITE                0x00000002
#define VM_EXEC                 0x00000004

// the control struct for a set of vma using the same Page Table
struct mm_struct {
    list_entry_t mmap_list;        // linear list link which sorted by start addr
of vma
    struct vma_struct *mmap_cache; // current accessed vma, used for speed
purpose
    pde_t *pgdir;                  // the Page Table of these vma
    int map_count;                 // the count of these vma
    void *sm_priv;                   // the private data for swap manager
};
```

除了以上内容，我们还需要为`vma_struct`和`mm_struct`定义和实现一些接口：包括它们的构造函数，以及如何把新的`vma_struct`插入到`mm_struct`对应的链表里。注意这两个结构体占用的内存空间需要用`kmalloc()`函数动态分配。

```
// kern/mm/vmm.c
// mm_create -  alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));

    if (mm != NULL) {
        list_init(&(mm->mmap_list));
        mm->mmap_cache = NULL;
        mm->pgdir = NULL;
        mm->map_count = 0;

        if (swap_init_ok) swap_init_mm(mm);//我们接下来解释页面置换的初始化
        else mm->sm_priv = NULL;
    }
    return mm;
}

// vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint_t vm_flags) {
    struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));
    if (vma != NULL) {
        vma->vm_start = vm_start;
        vma->vm_end = vm_end;
        vma->vm_flags = vm_flags;
    }
    return vma;
}
```

在插入一个新的`vma_struct`之前，我们要保证它和原有的区间都不重合。

```
// kern/mm/vmm.c
// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end);// next 是我们想插入的区间，这里顺便检验了
start < end
}
```

我们可以插入一个新的`vma_struct`，也可以查找某个虚拟地址对应的`vma_struct`是否存在。

```
// kern/mm/vmm.c

// insert_vma_struct -insert vma in mm's list link
void
insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
    assert(vma->vm_start < vma->vm_end);
    list_entry_t *list = &(mm->mmap_list);
    list_entry_t *le_prev = list, *le_next;

        list_entry_t *le = list;
        while ((le = list_next(le)) != list) {
            struct vma_struct *mmap_prev = le2vma(le, list_link);
            if (mmap_prev->vm_start > vma->vm_start) {
                break;
            }
            le_prev = le;
        }
    //保证插入后所有vma_struct按照区间左端点有序排列
    le_next = list_next(le_prev);

    /* check overlap */
    if (le_prev != list) {
        check_vma_overlap(le2vma(le_prev, list_link), vma);
    }
    if (le_next != list) {
        check_vma_overlap(vma, le2vma(le_next, list_link));
    }

    vma->vm_mm = mm;
    list_add_after(le_prev, &(vma->list_link));

    mm->map_count ++;//计数器
}

// find_vma - find a vma  (vma->vm_start <= addr <= vma_vm_end)
//如果返回NULL，说明查询的虚拟地址不存在/不合法，既不对应内存里的某个页，也不对应硬盘里某个可以
换进来的页
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;
```

```
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
                bool found = 0;
                list_entry_t *list = &(mm->mmap_list), *le = list;
                while ((le = list_next(le)) != list) {
                    vma = le2vma(le, list_link);
                    if (vma->vm_start<=addr && addr < vma->vm_end) {
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    vma = NULL;
                }
        }
        if (vma != NULL) {
            mm->mmap_cache = vma;
        }
    }
    return vma;
}
```

如果此时发生Page Fault怎么办？我们可以回顾之前的异常处理部分的知识。我们的`trapFrame`传递了`badvaddr`给`do_pgfault()`函数，而这实际上是`stval`这个寄存器的数值（在旧版的RISCV标准里叫做`sbadvaddr`），这个寄存器存储一些关于异常的数据，对于`PageFault`它存储的是访问出错的虚拟地址。

```
// kern/trap/trap.c
static int pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;//当前使用的mm_struct的指针，在vmm.c定义
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->cause, tf->badvaddr);
    }
    panic("unhandled page fault.\n");
}
// kern/mm/vmm.c
struct mm_struct *check_mm_struct;

// check_pgfault - check correctness of pgfault handler
static void
check_pgfault(void) {
    /* ...... */
    check_mm_struct = mm_create();
    /* ...... */
}
```

`do_pgfault()`函数在`vmm.c`定义，是页面置换机制的核心。如果过程可行，没有错误值返回，我们就可对页表做对应的修改，通过加入对应的页表项，并把硬盘上的数据换进内存，这时还可能涉及到要把内存里的一个页换出去，而`do_pgfault()`函数就实现了这些功能。如果你对这部分内容相当了解的话，那可以说你对于页面置换机制的掌握程度已经很棒了。

```
// kern/mm/vmm.c
int do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    //addr: 访问出错的虚拟地址
```

```
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);
    //我们首先要做的就是在mm_struct里判断这个虚拟地址是否可用
    pgfault_num++;
    //If the addr is not 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;
    }

    /* IF (write an existed addr ) OR
     *    (write an non_existed addr && addr is writable) OR
     *    (read  an non_existed addr && addr is readable)
     * THEN
     *    continue process
     */
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE); //按照页面大小把地址对齐

    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 {
        /*
        * Now we think this pte is a  swap entry, we should load data from disk
        * to a page with phy addr,
        * and map the phy addr with logical addr, trigger swap manager to record
        * the access situation of this page.
        *
        *    swap_in(mm, addr, &page) : alloc a memory page, then according to
        * the swap entry in PTE for addr, find the addr of disk page, read the
        * content of disk page into this memroy page
        *     page_insert :  build the map of phy addr of an Page with the
virtual addr la
        *   swap_map_swappable :  set the page swappable
        */
        if (swap_init_ok) {
            struct Page *page = NULL;
            //在swap_in()函数执行完之后，page保存换入的物理页面。
            //swap_in()函数里面可能把内存里原有的页面换出去
            swap_in(mm, addr, &page);  //(1) According to the mm AND addr, try
                                       //to load the content of right disk page
                                       //into the memory which page managed.
```

```
            page_insert(mm->pgdir, page, addr, perm); //更新页表，插入新的页表项
            //(2) According to the mm, addr AND page,
            // setup the map of phy addr <---> virtual addr
            swap_map_swappable(mm, addr, page, 1);  //(3) make the page
swappable.
            //标记这个页面将来是可以再换出的
            page->pra_vaddr = addr;
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed;
        }
   }

   ret = 0;
failed:
    return ret;
}
```

接下来我们看看FIFO页面置换算法是怎么在ucore里实现的。

FIFO页面置换算法

所谓FIFO(First in, First out)页面置换算法，相当简单，就是把所有页面排在一个队列里，每次换入页面的时候，把队列里最靠前（最早被换入）的页面置换出去。

换出页面的时机相对复杂一些，针对不同的策略有不同的时机。ucore 目前大致有两种策略，即积极换出策略和消极换出策略。

- **积极换出策略是指操作系统周期性地（或在系统不忙的时候）主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；**
- **消极换出策略是指只有当试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。**

目前的框架支持第二种情况，在`alloc_pages()`里面，如果此时试图得到空闲页且没有空闲的物理页时，我们才尝试换出页面到硬盘上。

```
// kern/mm/pmm.c
// alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE memory
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);
        //如果有足够的物理页面，就不必换出其他页面
        //如果n>1，说明希望分配多个连续的页面，但是我们换出页面的时候并不能换出连续的页面
         //swap_init_ok标志是否成功初始化了
        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        extern struct mm_struct *check_mm_struct;
        swap_out(check_mm_struct, n, 0);//调用页面置换的”换出页面“接口。这里必有n=1
    }
```

```
    return page;
}
```

类似`pmm_manager`，我们定义`swap_manager`，组合页面置换需要的一些函数接口。

```
// kern/mm/swap.h

struct swap_manager
{
     const char *name;
     /* Global initialization for the swap manager */
     int (*init)            (void);
     /* Initialize the priv data inside mm_struct */
     int (*init_mm)         (struct mm_struct *mm);
     /* Called when tick interrupt occured */
     int (*tick_event)      (struct mm_struct *mm);
     /* Called when map a swappable page into the mm_struct */
     int (*map_swappable)   (struct mm_struct *mm, uintptr_t addr, struct Page
*page, int swap_in);
     /* When a page is marked as shared, this routine is called to
      * delete the addr entry from the swap manager */
     int (*set_unswappable) (struct mm_struct *mm, uintptr_t addr);
     /* Try to swap out a page, return then victim */
     int (*swap_out_victim) (struct mm_struct *mm, struct Page **ptr_page, int
in_tick);
     /* check the page relpacement algorithm */
     int (*check_swap)(void);
};
```

我们来看`swap_in()`，`swap_out()`如何换入/换出一个页面.注意我们对物理页面的`Page`结构体做了一些改动。

```
// kern/mm/memlayout.h
struct Page {
    int ref;                        // page frame's reference counter
    uint_t flags;                 // array of flags that describe the status of
the page frame
    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)
};

// kern/mm/swap.c
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     struct Page *result = alloc_page();//这里alloc_page()内部可能调用swap_out()
     //找到对应的一个物理页面
     assert(result!=NULL);

     pte_t *ptep = get_pte(mm->pgdir, addr, 0);//找到/构建对应的页表项
    //将物理地址映射到虚拟地址是在swap_in()退出之后，调用page_insert()完成的
     int r;
```

```
     if ((r = swapfs_read((*ptep), result)) != 0)//将数据从硬盘读到内存
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n",
(*ptep)>>8, addr);
     *ptr_result=result;
     return 0;
}
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 *page;
          int r = sm->swap_out_victim(mm, &page, in_tick);//调用页面置换算法的接口
          //r=0表示成功找到了可以换出去的页面
         //要换出去的物理页面存在page里
          if (r != 0) {
                  cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                  break;
          }

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

          v=page->pra_vaddr;//可以获取物理页面对应的虚拟地址
          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) {
                      //尝试把要换出的物理页面写到硬盘上的交换区，返回值不为0说明失败了
                    cprintf("SWAP: failed to save\n");
                    sm->map_swappable(mm, v, page, 0);
                    continue;
          }
          else {
              //成功换出
                    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);
          }
          //由于页表改变了，需要刷新TLB
          //思考：  swap_in()的时候插入新的页表项之后在哪里刷新了TLB?
          tlb_invalidate(mm->pgdir, v);
     }
     return i;
}
```

`kern/mm/swap.c` 里其他的接口基本都是简单调用 `swap_manager` 的具体实现。值得一提的是 `swap_init()` 初始化里做的工作。

```
// kern/mm/swap.c
static struct swap_manager *sm;
```

```
int swap_init(void)
{
     swapfs_init();

     // Since the IDE is faked, it can only store 7 pages at most to pass the
test
     if (!(7 <= max_swap_offset &&
        max_swap_offset < MAX_SWAP_OFFSET_LIMIT)) {
        panic("bad max_swap_offset %08x.\n", max_swap_offset);
     }

     sm = &swap_manager_fifo;//use first in first out Page Replacement Algorithm
     int r = sm->init();

     if (r == 0)
     {
          swap_init_ok = 1;
          cprintf("SWAP: manager = %s\n", sm->name);
          check_swap();
     }

     return r;
}

int swap_init_mm(struct mm_struct *mm)
{
     return sm->init_mm(mm);
}

int swap_tick_event(struct mm_struct *mm)
{
     return sm->tick_event(mm);
}

int swap_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page,
int swap_in)
{
     return sm->map_swappable(mm, addr, page, swap_in);
}

int swap_set_unswappable(struct mm_struct *mm, uintptr_t addr)
{
     return sm->set_unswappable(mm, addr);
}
```

`kern/mm/swap_fifo.h` 完成了FIFO置换算法最终的具体实现。我们所做的就是维护了一个队列（用链表实现）。

```
// kern/mm/swap_fifo.h
#ifndef __KERN_MM_SWAP_FIFO_H__
#define __KERN_MM_SWAP_FIFO_H__

#include <swap.h>
extern struct swap_manager swap_manager_fifo;
```

```
#endif
// kern/mm/swap_fifo.c
/*  Details of FIFO PRA
 * (1) Prepare: In order to implement FIFO PRA, we should manage all swappable
pages, so we can
 * link these pages into pra_list_head according the time order. At first you
should
 * be familiar to the struct list in list.h. struct list is a simple doubly
linked list
 * implementation. You should know howto USE: list_init,
list_add(list_add_after),
 * list_add_before, list_del, list_next, list_prev. Another tricky method is to
transform
 *  a general list struct to a special struct (such as struct page). You can find
some MACRO:
 * le2page (in memlayout.h), (in future labs: le2vma (in vmm.h), le2proc (in
proc.h),etc.
 */

list_entry_t pra_list_head;
/*
 * (2) _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr
of pra_list_head.
 *     Now, From the memory control struct mm_struct, we can access FIFO PRA
 */
static int
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     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
_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;
}
/*
 *  (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
```

```
_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;
}
static int _fifo_init(void)//初始化的时候什么都不做
{
    return 0;
}

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

static int _fifo_tick_event(struct mm_struct *mm)//时钟中断的时候什么都不做
{ return 0; }


struct swap_manager swap_manager_fifo =
{
     .name            = "fifo swap manager",
     .init            = &_fifo_init,
     .init_mm         = &_fifo_init_mm,
     .tick_event      = &_fifo_tick_event,
     .map_swappable   = &_fifo_map_swappable,
     .set_unswappable = &_fifo_set_unswappable,
     .swap_out_victim = &_fifo_swap_out_victim,
     .check_swap      = &_fifo_check_swap,
};
```

我们通过`_fifo_check_swap()`,`check_swap()`,`check_vma_struct()`,`check_pgfault()`等接口对页面置换机制进行了简单的测试。具体测试的细节在这里不进行展示，同学们可以尝试自己进行测试。至此，实验三中的主要工作描述完毕，接下来请同学们完成本次实验练习。

实验报告要求

从oslab网站上取得实验代码后，进入目录labcodes/lab3，完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录labcodes/lab3下存放实验报告，推荐用**markdown**格式。每个小组建一个gitee或者github仓库，对于lab3中编程任务，完成编写之后，再通过git push命令把代码和报告上传到仓库。最后请一定提前或按时提交到git网站。

注意有“LAB3”的注释，代码中所有需要完成的地方（challenge除外）都有“LAB3”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

答辩提问

do_pgfault的完整流程

```
int
do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    int ret = -E_INVAL; // 初始化返回值，默认设置为无效地址错误
    // 查找包含地址 addr 的 vma
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++; // 增加页故障计数
    // 如果地址不在任何 vma 的范围内，则认为地址无效
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and can not find it in vma\n", addr);
        goto failed; // 跳到错误处理
    }

    /* 如果：
     * - 写入已存在的地址，或者
     * - 写入一个不存在的地址，并且该地址可写，或者
     * - 读取一个不存在的地址，并且该地址可读
     * 则继续处理页故障
     */
    uint32_t perm = PTE_U; // 基础权限为用户访问权限
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W); // 如果 vma 有写权限，添加读写权限
    }
    addr = ROUNDDOWN(addr, PGSIZE); // 将地址对齐到页大小

    ret = -E_NO_MEM; // 如果后续操作失败，默认返回内存不足错误

    pte_t *ptep = NULL;
    // 尝试找到页表项，如果页表不存在，则创建页表
    ptep = get_pte(mm->pgdir, addr, 1);
    if (*ptep == 0) { // 如果页表项为空，表示页面未分配
        // 分配一个新的物理页面并建立地址映射
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else { // 页表项不是空，可能是一个交换条目
        if (swap_init_ok) { // 检查是否已经初始化了页面置换
            struct Page *page = NULL;

            // (1) 根据 mm 和 addr，将正确的磁盘页面加载到内存中
            int result = swap_in(mm, addr, &page); // 调用 swap_in 函数从磁盘加载页面
            if (result != 0) {
                cprintf("swap_in failed\n");
                goto failed; // 如果加载失败，跳到错误处理
```

```
            }

            // (2) 根据 mm、addr 和 page 设置物理地址与逻辑地址之间的映射
            if (page_insert(mm->pgdir, page, addr, perm) != 0) {
                cprintf("page_insert failed\n");
                goto failed; // 如果插入失败，跳到错误处理
            }

            // (3) 将页面设置为可交换
            if (swap_map_swappable(mm, addr, page, 1) != 0) {
                cprintf("swap_map_swappable failed\n");
                goto failed; // 如果设置为可交换失败，跳到错误处理
            }

            page->pra_vaddr = addr; // 设置页面的虚拟地址为 addr
        } else {
            cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
            goto failed; // 如果页面置换未初始化且页表项不为空，跳到错误处理
        }
    }

    ret = 0; // 成功处理页故障
failed:
    return ret; // 返回处理结果
}
```

1. **查找包含地址 addr 的VMA** → 找不到 → 返回错误。
2. **设置页面权限。**
3. **对齐地址。**
4. 获取页表项：
    - 找不到页表项 → 分配页表项 → 如果失败 → 返回错误。
5. 页表项为空：
    - 分配新页面并映射 → 如果失败 → 返回错误。
6. 页表项不为空：
    - **从交换区换入页面** → 加载失败 → 返回错误。
    - **建立新映射** → 失败 → 返回错误。
    - **设置页面为可换出** → 失败 → 返回错误。
7. **返回成功或失败。**

FIFO换入换出具体是怎么实现的

FIFO 页面换入

函数 `_fifo_map_swappable` 负责处理页面的换入，即将一个页面标记为可替换。它的具体实现如下：

```
c复制代码static int
_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;  // 获取 FIFO 链表头节点
    list_entry_t *entry = &(page->pra_page_link);      // 获取页面对应的链表节点

    assert(entry != NULL && head != NULL);  // 确保节点不为空

    // 将新页面插入到 pra_list_head 队列的末尾（链表的头部表示队列的末尾）
    list_add(head, entry);

    return 0;
}
```

换入的实现逻辑

1. **链表初始化**：每当一个页面变得"可换出"时，FIFO 算法会将它添加到一个页面置换链表 `pra_list_head` 中。
2. **插入页面**：使用 `list_add()` 将页面插入链表的末尾，表示这个页面是最新加载的。如果未来发生缺页，需要进行页面置换，那么最早进入的页面就会首先被替换掉。

通过这一过程，`_fifo_map_swappable` 将所有可以被替换的页面加入到页面队列中，并保持顺序。最新加载的页面在链表末尾，最早加载的页面在链表头部。

FIFO 页面换出

函数 `_fifo_swap_out_victim` 负责处理页面的换出，即选择一个页面从内存中移除。它的具体实现如下：

```
c复制代码static int
_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;  // 获取链表头节点
    assert(head != NULL);
    assert(in_tick == 0);  // 确保当前不是在时钟中断期间

    // 选择最早到达的页面，即 pra_list_head 链表中的最后一个节点（队列头）
    list_entry_t* entry = list_prev(head);
    if (entry != head) {
        // 从队列中删除该页面，并将页面指针赋值给 ptr_page
        list_del(entry);
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        *ptr_page = NULL;  // 如果队列为空，设置 ptr_page 为 NULL
    }

    return 0;
```

```
}
```

换出的实现逻辑

1. **链表头**：通过 `mm->sm_priv` 获取链表头节点，链表用于记录所有可替换的页面。
2. **选择换出页面**：通过 `list_prev(head)` 获取链表中最早到达的页面。由于链表头部（`head`）表示队列的末尾，`list_prev(head)` 会返回链表中最早到达的页面（即在逻辑上是队列的头部）。
3. **删除页面**：调用 `list_del(entry)` 将页面从链表中移除，表示该页面将被换出。
4. **更新换出指针**：通过 `le2page(entry, pra_page_link)` 获取页面结构体，并将其指针赋值给 `ptr_page`，以供后续处理使用。

FIFO 页面换入和换出的总结

- 换入（页面变为可换出）：
  - 使用 `_fifo_map_swappable()` 将页面添加到 `pra_list_head` 链表的末尾。链表记录了所有可替换的页面，最新到达的页面在链表末尾。
- 换出（页面置换）：
  - 使用 `_fifo_swap_out_victim()` 从 `pra_list_head` 链表中选择最早到达的页面（队列头部）进行换出。
  - 被换出的页面从链表中移除，以便释放内存或将页面写入磁盘。

总结

FIFO 算法通过维护一个按时间顺序排列的页面链表实现页面的换入换出：

- **换入时**，将页面插入到链表的末尾，表示它是最新到达的页面。
- **换出时**，从链表头部选择最早到达的页面进行换出。

clock的完整流程

1. Clock 页面置换的整体思想

Clock 算法通过模拟一个时钟指针来遍历所有页面，用于确定哪个页面应当被换出。它会循环检查页面的访问位，如果某个页面的访问位为 0，则选择它进行替换；否则，将访问位置为 0，继续下一个页面，直到找到可以替换的页面为止。

2. Clock 算法的主要函数和流程

2.1 初始化

`_clock_init_mm(struct mm_struct *mm)`

- **作用**：初始化页面链表和时钟指针。
- 步骤：
  1. 使用 `list_init(&pra_list_head)` 初始化页面置换链表 `pra_list_head`。
  2. 将 `curr_ptr` 指针初始化为 `pra_list_head`，即当前替换位置指向链表的头节点。
  3. 让 `mm` 的 `sm_priv` 指向 `pra_list_head`，以便在内存管理中能够访问页面链表。

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

2.2 页面换入

`_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)`

- **作用**：将一个页面标记为可换出，并添加到链表中。
- 步骤：
    1. 获取页面的链表节点 `entry`。
    2. 使用 `list_add_before(&pra_list_head, entry)` 将页面插入到链表末尾，这样页面按顺序被访问。
    3. 将页面的访问位 `page->visited` 置为 1，表示页面刚刚被访问过。

```
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);
    assert(entry != NULL && curr_ptr != NULL);
    list_add_before(&pra_list_head, entry);
    page->visited = 1;
    return 0;
}
```

2.3 页面换出

`_clock_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)`

- **作用**：根据 Clock 算法选择一个页面进行替换。
- 步骤：
    1. 获取链表的头节点。
    2. 开始遍历链表，使用时钟指针 `curr_ptr` 从当前页面开始向前移动。
    3. 如果 `curr_ptr` 回到链表头部，则继续从下一个页面开始。
    4. 获取当前页面指针 `page`。
    5. 如果页面的访问位

        ```
        visited
        ```

        为 0，则选定此页面为被替换页面：
        - 使用 `list_del(curr_ptr)` 将该页面从链表中移除。
        - 将页面指针赋值给 `ptr_page`。
    6. 如果页面已被访问（`visited` 为 1），则将 `visited` 置为 0，表示页面已经被再次访问。

```
```

```
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) {
        curr_ptr = list_next(curr_ptr);
        if (curr_ptr == &pra_list_head) {
            curr_ptr = list_next(curr_ptr);
        }

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

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

3. Clock 页面置换的特点和优缺点

- **访问位的利用**：Clock 算法通过页面的 `visited` 访问位来决定是否换出页面。当一个页面被访问时，`visited` 会被置为 1，当时钟指针再次经过时，它会检查页面是否已经被访问过。如果访问位为 1，则表示该页面最近被访问过，不能被换出，需要将访问位重置为 0，继续寻找其他页面。
- **循环遍历**：Clock 算法不断循环遍历页面链表，直到找到合适的页面。它类似于一个“时钟”，通过指针不断前移寻找换出页面，因此也叫“Clock”算法。
- **优点**：
  - **效率较高**：相比于简单的 FIFO 算法，Clock 算法通过引入访问位，能够更好地决定换出哪个页面，减少不必要的页面替换。
- **缺点**：
  - **近似于 LRU**：虽然 Clock 算法改进了 FIFO，接近 LRU（最近最少使用），但它并不是真正的 LRU，可能还是会替换掉一些比较常用的页面。

4. 总结

- **初始化**：`_clock_init_mm` 函数初始化页面链表和时钟指针。
- **页面换入**：当一个页面变得可替换时，使用 `_clock_map_swappable` 函数将页面加入链表末尾，并将访问位 `visited` 置为 1。
- **页面换出**：使用 `_clock_swap_out_victim` 函数来选择需要被换出的页面。它会根据访问位来决定是否换出当前页面，访问位为 0 时，页面被换出；如果访问位为 1，则将访问位清零，继续查找下一个页面。

整个 Clock 页面置换流程通过时钟指针来遍历页面，依次检查每个页面的访问情况，以决定哪个页面可以被替换，从而达到较好的页面置换效果。

LRU的具体流程

1. LRU 算法整体思想

LRU 算法的基本思想是：当一个新的页面需要进入内存时，会将最近最少使用的页面置换出去。在实现中，通常使用一个链表来跟踪页面的使用顺序，链表尾部是最新访问的页面，链表头部是最久未访问的页面。

在这段代码中，`lru_swap` 使用了双向链表来实现 LRU 算法，其中 `pra_list_head2` 是链表的头节点，`curr_ptr2` 是一个当前指针，用于指向当前被访问或替换的页面。

2. LRU 算法的主要函数及其具体流程

2.1 初始化

`_lru_init_mm(struct mm_struct *mm)`

- **作用**：初始化页面置换的链表和指针。
- 步骤：
  1. 调用 `list_init(&pra_list_head2)` 初始化页面置换链表 `pra_list_head2`。
  2. 将当前指针 `curr_ptr2` 初始化为指向链表头 `pra_list_head2`。
  3. 将 `mm` 的 `sm_priv` 指向 `pra_list_head2`，这样 `mm_struct` 中可以直接访问页面置换链表。

```
static int _lru_init_mm(struct mm_struct *mm) {
    list_init(&pra_list_head2);
    curr_ptr2 = &pra_list_head2;
    mm->sm_priv = &pra_list_head2;
    return 0;
}
```

2.2 页面换入

`_lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)`

- **作用**：将新页面插入页面链表中，以便记录页面的访问顺序。
- 步骤：
  1. 获取页面的链表节点 `entry`。
  2. 获取页面链表的头节点 `head`。
  3. 将页面 `page` 插入到链表末尾，这样新的页面始终处于链表末尾，表示最近被访问过。

```
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);
    list_entry_t *head = &pra_list_head2;
    assert(entry != NULL && head != NULL);
    list_add(head, entry); // 将新页面插入链表的末尾
    return 0;
}
```

2.3 页面换出

`_lru_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)`

- **作用**：选择需要被换出的页面。
- 步骤：
    1. 获取页面链表的头节点 `head`。
    2. 获取链表中的第一个页面（即最久未被访问的页面）。
    3. 如果页面存在，则从链表中移除并将页面指针赋值给 `ptr_page`。

```
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 *entry = list_prev(head); // 获取最久未被访问的页面（链表头）
    if (entry != head) {
        list_del(entry); // 从链表中移除
        *ptr_page = le2page(entry, pra_page_link);
    } else {
        *ptr_page = NULL;
    }
    return 0;
}
```

2.4 页面访问更新

`_lru_access(uintptr_t addr)`

- **作用**：更新页面的访问顺序。
- 步骤：
    1. 获取页面对应的 `Page` 结构体。
    2. 如果页面存在，则将其从链表中删除，再重新插入链表末尾，表示页面最近被访问过。

```
static uintptr_t _lru_access(uintptr_t addr) {
    list_entry_t *head = &pra_list_head2;
    pte_t **ptep_store = NULL;
    struct Page *page = get_page(boot_pgdir, addr, ptep_store);
    if (page != NULL) {
        list_entry_t *entry = &(page->pra_page_link);
        list_del(entry); // 删除旧位置
        list_add(head, entry); // 插入到链表末尾
    }
    return addr;
}
```

3. LRU 页面置换流程总结

- **初始化**：调用 `_lru_init_mm` 初始化页面链表，设置当前指针为链表头。
- **页面换入**：当一个页面变得可换出时，调用 `_lru_map_swappable` 将页面加入链表末尾，表示页面被访问过。
- **页面换出**：调用 `_lru_swap_out_victim` 从链表头选择一个页面进行置换，链表头部的页面是最久未被访问的。
- **页面访问更新**：每次访问页面时调用 `_lru_access` 更新页面在链表中的位置，将其移动到链表末尾，表示最近被访问。

4. 代码的优缺点

- 优点：
  - **近似 LRU**：该实现通过一个链表记录页面的访问顺序，链表末尾的页面是最近访问过的，链表头部的页面是最久未被访问的，这样实现了 LRU 的思想。
- 缺点：
  - **性能问题**：在每次访问页面时，需要将页面从链表中删除并插入到链表末尾，操作链表的时间复杂度为 O(N)，当页面数量较多时，性能可能不佳。
  - **空间开销**：使用链表管理页面需要额外的内存空间，尤其是对于大量页面的情况，链表节点会带来额外的存储开销。

5. 总结

LRU 算法通过链表来管理页面的访问顺序，每次页面被访问时，将其移到链表的末尾。当需要置换页面时，从链表头选择页面进行换出。这种实现方式能够较好地逼近 LRU 的置换策略，适用于需要较高页面命中率的场景。

alloc_page里面有个n>1,break;是什么作用?

处理缺页异常是一页一页处理的。

get pte的每一行都提问了

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create) {
    /*
     * 如果需要访问一个物理地址，请使用 KADDR() 宏。
     * 请阅读 pmm.h 以了解有用的宏。
     *
     * 可能您需要帮助注释，下面的注释可以帮助您完成代码。
     *
     * 以下是一些有用的宏和定义，您可以在下面的实现中使用它们。
     * 宏或函数：
     *   PDX(la) = 获取虚拟地址 la 的页目录项的索引。
     *   KADDR(pa): 传入一个物理地址并返回对应的内核虚拟地址。
     *   set_page_ref(page, 1): 表示此页面被引用了一次。
     *   page2pa(page): 获取此（struct Page *）page 管理的内存的物理地址。
```

```
     *   struct Page * alloc_page(): 分配一个页面。
     *   memset(void *s, char c, size_t n): 将指针 s 指向的内存区域的前 n 个字节设置为
指定值 c。
     * 定义:
     *   PTE_P 0x001 - 页表/页目录项标志位：存在（Present）
     *   PTE_W 0x002 - 页表/页目录项标志位：可写（Writeable）
     *   PTE_U 0x004 - 页表/页目录项标志位：用户可访问（User can access）
     */

    // 获取一级页目录项指针
    pde_t *pdep1 = &pgdir[PDX1(la)];

    // 如果一级页目录项不包含有效页表
    if (!(*pdep1 & PTE_V)) {
        struct Page *page;

        // 如果没有允许创建，或者分配新页面失败，则返回 NULL
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }

        // 将该页面的引用计数设置为 1，表示页面被引用了一次
        set_page_ref(page, 1);

        // 获取页面的物理地址
        uintptr_t pa = page2pa(page);

        // 将分配的物理页面对应的内核虚拟地址清零，大小为 PGSIZE
        memset(KADDR(pa), 0, PGSIZE);

        // 设置一级页目录项，指向分配的页面，设置权限为用户访问和有效
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }

    // 获取二级页目录项指针，使用一级页目录项获取二级页目录的内核虚拟地址
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];

    // 如果二级页目录项不包含有效页表
    if (!(*pdep0 & PTE_V)) {
        struct Page *page;

        // 如果没有允许创建，或者分配新页面失败，则返回 NULL
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }

        // 将该页面的引用计数设置为 1，表示页面被引用了一次
        set_page_ref(page, 1);

        // 获取页面的物理地址
        uintptr_t pa = page2pa(page);

        // 将分配的物理页面对应的内核虚拟地址清零，大小为 PGSIZE
        memset(KADDR(pa), 0, PGSIZE);

        // 设置二级页目录项，指向分配的页面，设置权限为用户访问和有效
```

```
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
    }

    // 最终返回三级页表项的内核虚拟地址
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

关于三级页表的问题

```
// 线性地址 'la' 的四部分结构如下：
//
// +--------9-------+-------9--------+-------9--------+---------12----------+
// | Page Directory | Page Directory |   Page Table   | Offset within Page  |
// |     Index 1    |    Index 2     |                |                     |
// +----------------+----------------+----------------+---------------------+
//  \-- PDX1(la) --/ \-- PDX0(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \------------------PPN(la)----------------------/
//
// 宏 `PDX1`、`PDX0`、`PTX`、`PGOFF` 和 `PPN` 用于分解线性地址。
// 要通过 `PDX(la)`、`PTX(la)` 和 `PGOFF(la)` 构造线性地址 `la`，请使用
`PGADDR(PDX(la), PTX(la), PGOFF(la))`。

// RISC-V 使用 39 位的虚拟地址访问 56 位的物理地址！
// Sv39 虚拟地址：
// +----9----+----9---+----9---+---12--+
// |  VPN[2] | VPN[1] | VPN[0] | PGOFF |
// +---------+----+---+--------+-------+
//
// Sv39 物理地址：
// +----26---+----9---+----9---+---12--+
// |  PPN[2] | PPN[1] | PPN[0] | PGOFF |
// +---------+----+---+--------+-------+
//
// Sv39 页表项：
// +----26---+----9---+----9---+---2----+-------8-------+
// |  PPN[2] | PPN[1] | PPN[0] |Reserved|D|A|G|U|X|W|R|V|
// +---------+----+---+--------+--------+---------------+
```

有两个汇编文件，出现页错误的虚拟地址是怎么得到的?就在汇编文件里有

在汇编文件中，页错误的虚拟地址是通过访问 `stval` 寄存器得到的。在 RISC-V 架构中，`stval`（在旧版标准中称为 `sbadvaddr`）寄存器用于存储产生异常时的附加信息，当发生页错误时，它存储导致页错误的虚拟地址。

在你提供的汇编代码中，页错误的虚拟地址是通过以下操作获取的：

```
csrr s3, 0x143 // 访问寄存器 stval，用于记录错误虚拟地址（旧版称为 sbadvaddr）
```

- `csrr s3, 0x143`：`csrr` 指令用于从控制状态寄存器读取值到通用寄存器中。在这里，`0x143` 是 `stval` 寄存器的地址，`s3` 是目标通用寄存器。因此，这条指令的作用是将 `stval` 的值（即发生页错误的虚拟地址）存储到寄存器 `s3` 中。

随后，在 `SAVE_ALL` 宏中，将寄存器 `s3` 的值保存到栈中：

```
STORE s3, 34*REGBYTES(sp)
```

这样，当陷阱处理程序需要处理页错误时，它可以通过 `s3` 的值或通过栈中保存的值，获取到导致页错误的虚拟地址。

在整个流程中：

1. 当发生页错误时，硬件会将导致页错误的虚拟地址存储到 `stval` 寄存器中。
2. 在 `SAVE_ALL` 宏中，通过 `csrr s3, 0x143` 从 `stval` 寄存器读取虚拟地址，并将其保存到寄存器 `s3`。
3. 接着将 `s3` 的值保存到栈中，以便在处理陷阱时使用。

因此，页错误的虚拟地址是通过读取 `stval` 寄存器（通过 `csrr` 指令）来获取的。

发生缺页异常时，储存导致产生异常的虚拟地址是哪个寄存器?

发生缺页异常（Page Fault）时，导致产生异常的虚拟地址存储在 **stval** 寄存器中（对于 RISC-V 架构）。

在 RISC-V 架构中，当发生页面错误等异常时，**stval** 寄存器会保存产生异常的虚拟地址。这个寄存器的值会被传递给 `do_pgfault()` 函数进行处理，以便操作系统能够确定哪个虚拟地址导致了缺页异常并采取适当的操作。

在代码中，可以看到异常处理程序 `exception_handler` 中使用了 `tf->badvaddr`，而这个 `badvaddr` 实际上对应的就是 **stval** 寄存器的值。

assert函数的作用

断言检查表达式的真假，用于检查缺页异常的处理

怎么判断插入vma到mm的地址合法性应用?

在插入一个新的`vma_struct`之前，我们要保证它和原有的区间都不重合。

```
// kern/mm/vmm.c
// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end);// next 是我们想插入的区间，这里顺便检验了
start < end
}
```

请你用一句话总结Lab3?

处理缺页异常

模拟的交换区一扇磁盘的大小

是512字节，一页4096字节，一页对应8个扇区。

belady异常是什么?

为进程分配的物理块数增大时，缺页次数不减反增的异常现象。

get pte里面memset的作用，kaddr的作用

`memset` 的作用

在 `get_pte` 函数中，`memset` 的作用是对新分配的页表进行初始化。代码如下：

```
memset(KADDR(pa), 0, PGSIZE);
```

具体来说，`get_pte` 函数会检查页表是否已经存在，如果页表不存在，就会分配一个新的物理页作为页表。在分配新的页表后，需要将该页表的内容清零。这是因为新分配的内存可能包含之前的垃圾数据。清零操作可以确保新页表中所有的页表项初始状态为 0，即都为空，不存在任何映射关系。这样可以保证页表的正确性，避免由于存在错误的数据而引发不期望的行为。

`KADDR` 的作用

`KADDR` 的作用是将物理地址转换为对应的内核虚拟地址。具体来说，在分配新的页表后，`page2pa(page)` 得到的是该页表的物理地址，但直接通过物理地址进行访问在编程中是不便的。在操作系统内核中，我们往往通过虚拟地址来访问内存。因此需要使用 `KADDR` 宏将物理地址转换为内核虚拟地址，才能使用这个地址进行进一步的操作，例如 `memset` 清零。

例如：

```
uintptr_t pa = page2pa(page);
memset(KADDR(pa), 0, PGSIZE);
```

在这段代码中：

- `page2pa(page)` 返回新分配页表的物理地址。
- `KADDR(pa)` 将物理地址转换为内核可以直接访问的虚拟地址。
- `memset(KADDR(pa), 0, PGSIZE)` 将页表所在的页的所有字节设置为 0。

这样，内核可以通过虚拟地址来操作物理内存，而不需要直接处理物理地址。

总结

- `memset` 的作用是初始化新分配的页表，将其内容清零，确保页表项初始状态为空。
- `KADDR` 的作用是将物理地址转换为内核虚拟地址，以便内核可以通过虚拟地址访问物理内存。这样操作系统可以更加方便地操作页表和其他内存结构。

vma和mm结构体，现场找出来，告诉有什么成员变量，作用是什么，关系是?

```
struct vma_struct {
    struct mm_struct *vm_mm; // 指向使用相同页目录表 (PDT) 的 vma 集合
    uintptr_t vm_start;      // vma 的起始地址
    uintptr_t vm_end;        // vma 的结束地址，不包含 `vm_end` 本身
    uint_t vm_flags;         // vma 的标志，描述权限（如可读、可写、可执行等）
    // `vm_flags` 表示一段虚拟地址对应的权限，这些权限也会在页表项中进行相应的设置
    list_entry_t list_link;  // 用于将 vma 结构体按起始地址排序连接成线性链表
};
```

```
// 控制使用相同页目录表 (PDT) 的一组 vma 的结构体
struct mm_struct {
    list_entry_t mmap_list;        // 按 vma 起始地址排序的线性链表链接
    struct vma_struct *mmap_cache; // 当前访问的 vma，用于加快查找速度
    pde_t *pgdir;                  // 这些 vma 的页目录表 (PDT)
    int map_count;                 // 这些 vma 的数量
    void *sm_priv;                 // 用于页面置换管理器的私有数据
};
```

1. `vma_struct` **(虚拟内存区域)**

`vma_struct` 用于描述虚拟地址空间中的一个连续区域（例如代码段、数据段、堆或栈）。

成员变量及其作用：

- `struct mm_struct \*vm_mm`：指向包含该 `vma` 的 `mm_struct`，用于访问整个进程的地址空间结构。可以理解为该 `vma` 属于哪个地址空间。
- `uintptr_t vm_start`：该虚拟内存区域的起始地址。
- `uintptr_t vm_end`：该虚拟内存区域的结束地址（不包含 `vm_end` 本身）。
- `uint_t vm_flags`：描述该虚拟内存区域的权限和属性，例如可读、可写、可执行等。常用标志有：
  - `VM_READ`：可读
  - `VM_WRITE`：可写
  - `VM_EXEC`：可执行 这些权限标志也会在页表项中进行相应的设置。
- `list_entry_t list_link`：用于将多个 `vma_struct` 链接成线性链表，按 `vm_start` 地址排序。这使得可以遍历整个虚拟地址空间的所有 `vma`，从而方便查找某个地址所属的内存区域。

2. `mm_struct` **(内存管理结构体)**

`mm_struct` 用于描述整个进程的虚拟地址空间，包括代码段、数据段、堆、栈等所有虚拟内存区域。

成员变量及其作用：

- `list_entry_t mmap_list`：按 `vma` 起始地址排序的链表，用于链接该进程的所有 `vma_struct`。通过这个链表可以遍历该进程的所有虚拟内存区域。
- `struct vma_struct \*mmap_cache`：缓存当前访问的 `vma`，用于加快查找速度。在内存管理中，很多情况下会多次访问相同的 `vma`，因此用缓存来避免重复遍历链表，加快查找效率。
- `pde_t \*pgdir`：指向该进程的页目录表 (PDT)，用于管理该进程的页表结构。页目录表负责将虚拟地址映射到物理地址。
- `int map_count`：该进程拥有的 `vma` 的数量，用于统计该进程地址空间中包含多少个虚拟内存区域。
- `void \*sm_priv`：页面置换管理器的私有数据指针，用于与页面置换算法相关的数据。不同的页面置换算法可能需要不同的数据结构，这个指针可以指向相应的数据结构，例如 FIFO 链表或 LRU 队列等。

3. **`vma_struct` 和 `mm_struct` 的关系**

- `mm_struct` 描述了一个进程的整个虚拟地址空间，而 `vma_struct` 则描述该地址空间中的一个连续区域（例如代码段、数据段等）。
- `vm_mm` 成员：每个 `vma_struct` 通过 `vm_mm` 指针指向它所属的 `mm_struct`，这表示该虚拟内存区域属于哪个地址空间。
- `mmap_list` **链接**：`mm_struct` 通过 `mmap_list` 链接了该进程的所有 `vma_struct`，这些 `vma_struct` 通过 `list_link` 链接起来形成一个链表，按地址从低到高的顺序排列。
- **缓存机制**：为了加快访问速度，`mm_struct` 还包含了一个 `mmap_cache`，用于缓存最近访问的 `vma_struct`，这样在访问相邻内存区域时可以避免遍历整个链表。

LRU为什么没有Belady异常?

先进先出只和时间有关系，比如一个页面排在前面，被频繁访问，FIFO算法却不停替换它。LRU永久替换的都是最久未访问的。

为什么fifo会有belady

FIFO 算法的局限性在于它总是替换最早进入的页面，而不考虑页面在未来是否会被使用。这会导致一些频繁访问的页面被过早替换，进而导致更多的缺页。例如：

- 在上面的例子中，页面 `1` 和 `2` 在访问序列中被频繁访问，但在有 4 个页面帧时，这些页面被替换得更频繁，导致缺页次数增加。
- FIFO 没有考虑到页面的“局部性”，即某些页面在一段时间内会被频繁访问，而它仅仅根据页面进入内存的顺序来选择被替换的页面，因此更容易发生误换出。

重点关注这两个宏定义

```
// address in page table or page directory entry
//将pte与~0x3FF相与，清除低10位标志位，保留页帧号部分
#define PTE_ADDR(pte)   (((uintptr_t)(pte)& ~0x3FF)<<(PTXSHIFT - PTE_PPN_SHIFT))
//从页表项 pte 中提取页面的基地址，去掉低10位的标志位，并左移以得到物理地址中的页帧号字段
#define PDE_ADDR(pde)   PTE_ADDR(pde)
//从页目录项 pde 中提取页面的基地址
```

`#define PTE_ADDR(pte)   (((uintptr_t)(pte)& ~0x3FF)<<(PTXSHIFT . PTE_PPN_SHIFT))`

从页表项 pte 中提取页面的基地址，去掉低10位的标志位，并左移以得到物理地址中的页帧号字段

`#define PDE_ADDR(pde)   PTE_ADDR(pde)`

从页目录项 pde 中提取页面的基地址