实验三：虚拟内存管理

1 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

2 实验内容

做完实验二后，大家可以了解并掌握物理内存管理中的连续空间分配算法的具体实现以及如何建立二级页表。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FIFO页替换算法的实现，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。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extended clock页替换算法。

2.1 练习

练习0：填写已有实验

本实验依赖实验1/2。请把你做的实验1/2的代码填入本实验中代码中有“LAB1”,“LAB2”的注释相应部分。

练习1：给未被映射的地址映射上物理页（需要编程）

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。注意：在 LAB2 EXERCISE 1 处填写代码。执行“make qemu”后，如果通过 check_pgfault 函数的测试后，会有“check_pgfault() succeeded!”的输出，表示练习 1 基本正确。

练习2：补充完成基于FIFO的页面替换算法（需要编程）

完成vmm.c中的do_pgfault函数，并且在实现FIFO算法的swap_fifo.c中完成map_swappable和swap_out_vistim函数。通过对swap的测试。注意：在LAB2 EXERCISE 2处填写代码。执行“make qemu”后，如果通过check_swap函数的测试后，会有“check_swap() succeeded!”的输出，表示练习2基本正确。

扩展练习Challenge：实现识别dirty bit的extended clock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。(基本实验完成后一周内完成，单独提交）。

2.2 项目组成

表1： 实验二文件列表

```
|-- boot
|-- kern
|   |-- driver
|   |   |-- …
|   |   |-- ide.c
|   |   `-- ide.h
|   |-- fs
|   |   |-- fs.h
|   |   |-- swapfs.c
|   |   `-- swapfs.h
|   |-- init
```

```
|  |  |-- ...
|  |  `-- init.c
|  |-- mm
|  |  |-- default_pmm.c
|  |  |-- default_pmm.h
|  |  |-- memlayout.h
|  |  |-- mmu.h
|  |  |-- pmm.c
|  |  |-- pmm.h
|  |  |-- swap.c
|  |  |-- swap.h
|  |  |-- swap_fifo.c
|  |  |-- swap_fifo.h
|  |  |-- vmm.c
|  |  `-- vmm.h
|  |-- sync
|  `-- trap
|     |-- trap.c
|     `-- ...
|-- libs
|     |-- list.h
|     `-- ...
`-- tools
```

相对与实验二，实验三主要增加的文件如上表红色部分所示，主要修改的文件如上表紫色部分所示，其他需要用到的重要文件用黑色表示。主要改动如下：

- kern/mm/default_pmm.[ch]：实现基于struct pmm_manager类框架的Fist-Fit物理内存分配参考实现（分配最小单位为页，即4096字节），相关分配页和释放页等实现会间接被kmalloc/kfree等函数使用。
- kern/mm/pmm.[ch]：pmm.h定义物理内存分配类框架struct pmm_manager。 pmm.c包含了对此物理内存分配类框架的访问，以及与建立、修改、访问页表相关的各种函数实现。在本实验中会用到kmalloc/kfree等函数。
- libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用list.h中定义的函数。在本实验中会多次用到插入，删除等操作函数。
- kern/driver/ide.[ch]：定义和实现了内存页swap机制所需的磁盘扇区的读写操作支持；在本实验中会涉及通过swapfs_*函数间接使用文件中的函数。故了解即可。
- kern/fs/*：定义和实现了内存页swap机制所需从磁盘读数据到内存页和写内存数据到磁盘上去的函数 swapfs_read/swapfs_write。在本实验中会涉及使用这两个函数。
- kern/mm/memlayout.h：修改了struct Page，增加了两项pra_*成员结构，其中pra_page_link可以用来建立描述各个页访问情况（比如根据访问先后）的链表。在本实验中会涉及使用这两个成员结构，以及le2page等宏。
- kern/mm/vmm.[ch]：vmm.h描述了mm_struct，vma_struct等表述可访问的虚存地址访问的一些信息，下面会进一步详细讲解。vmm.c涉及mm,vma结构数据的创建/销毁/查找/插入等函数，这些函数在check_vma、check_vmm等中被使用，理解即可。而page fault处理相关的do_pgfault函数是本次实验需要涉及完成的。
- kern/mm/swap.[ch]：定义了实现页替换算法的类框架struct swap_manager。swap.c包含了对此页替换算法类框架的初始化、页换入/换出等各种函数实现。重点是要理解何时调用swap_out和swap_in函数。和如何在此框架下连接具体的页替换算法实现。check_swap函数以及被此函数调用的_fifo_check_swap函数完成了对本次实验中的练习2：FIFO页替换算法基本正确性的

检查，可了解，便于知道为何产生错误。

- kern/mm/swap_fifo.[ch]：FIFO页替换算法的基于类框架struct swap_manager的简化实现，主要被swap.c的相关函数调用。重点是_fifo_map_swappable函数（可用于建立页访问属性和关系，比如访问时间的先后顺序）和_fifo_swap_out_victim函数（可用于实现挑选出要换出的页），当然换出哪个页需要借助于fifo_map_swappable函数建立的某种属性关系，已选出合适的页。
- kern/mm/mmu.h：其中定义额也页表项的各种属性位，比如PTE_P\PET_D\PET_A等，对于实现扩展实验的clock算法会有帮助。

本次实验的主要练习集中在vmm.c中的do_pgfault函数和swap_fifo.c中的_fifo_map_swappable函数、_fifo_swap_out_victim函数。

编译执行

编译并运行代码的命令如下：

```
make
make qemu
```

则可以得到如附录所示的显示内容（仅供参考，不是标准答案输出）

3 虚拟内存管理

什么是虚拟内存？简单地说，是指程序员或CPU “需要”和直接“看到”的内存，这其实暗示了两点：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，那二者的地址一般不是相等的。通过操作系统的某种内存管理和映射技术可建立虚拟内存与实际的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转换为另外一个物理内存地址。

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直接“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，这种技术属于lazy load技术，简称按需分页（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”，我们称为内存空间虚拟化。

ucore在实现上述技术时，需要解决的三个关键问题是：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？

对于第一个问题的出现，在于实验二中有关内存的数据结构和相关操作都是直接针对实际存在的资源--物理内存空间的管理，没有从一般应用程序对内存的“需求”考虑，即需要有相关的数据

结构和操作来体现一般应用程序对虚拟内存的“需求”。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，ucore是通过page fault异常处理来间接完成这二者之间的衔接。

ucore通过建立mm_struct和vma_struct数据结构，描述了ucore模拟应用程序运行所需的合法内存空间。当访问内存产生page fault异常时，可获得访问的内存的方式（读或写）以及具体的虚拟内存地址，这样ucore就可以查询此地址，看是否属于vma_struct数据结构中描述的合法地址范围中，如果在，则可根据具体情况进行请求调页/页换入换出处理（这就是练习1涉及的部分）；如果不在，则报错。

在ucore中描述应用程序对虚拟内存“需求”的数据结构是vma_struct （定义在vmm.h中），以及针对vma_struct的函数操作。这里把一个vma_struct结构的变量简称为vma变量。vma_struct的定义如下：

```
struct vma_struct {
    // the set of vma using the same PDT
        struct mm_struct *vm_mm;
   uintptr_t vm_start;      // start addr of vma
   uintptr_t vm_end;        // end addr of vma
         uint32_t vm_flags;       // flags of vma
         //linear list link which sorted by start addr of vma
         list_entry_t list_link;
};
```

vm_start和vm_end描述了一个连续地址的虚拟内存空间的起始位置和结束位置，这两个值都应该是 PGSIZE 对齐的，而且描述的是一个合理的地址空间范围（即严格确保 vm_start < vm_end 的关系）；list_link是一个双向链表，按照从小到大的顺序把一系列用vma_struct表示的虚拟内存空间链接起来，并且还要求这些链起来的 vma_struct 应该是不相交的，即vma之间的地址空间无交集；vm_flags表示了这个虚拟内存空间的属性，目前的属性包括：

```
#define VM_READ       0x00000001 //只读
#define VM_WRITE      0x00000002 //可读写
#define VM_EXEC       0x00000004 //可执行
```

vm_mm是一个指针，指向一个比vma_struct更高的抽象层次的数据结构mm_struct，这里把一个mm_struct结构的变量简称为mm变量。这个数据结构表示了包含所有虚拟内存空间的共同属性，具体定义如下

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

mmap_list是双向链表头，链接了所有属于同一页目录表的虚拟内存空间，mmap_cache是指向当前正在使用的虚拟内存空间，由于操作系统执行的“局部性”原理，当前正在用到的虚拟内存空间在接下来的操作中可能还会用到，这时就不需要查链表，而是直接使用此指针就可找到下一次要用到的虚拟内存空间。由于 mmap_cache 的引入，可使得 mm_struct 数据结构的查询加速 30%以上。pgdir 所指向的就是 mm_struct 数据结构所维护的页表。通过访问pgdir可以查找某虚拟地址对应的页表项是否存在以及页表项的属性等。map_count记录 mmap_list 里面链接的 vma_struct 的个数。sm_priv指向用来链接记录页访问情况的链表头，这建立了mm_struct和后续要讲到的

swap_manager之间的联系。

涉及vma_struct的操作函数也比较简单，主要包括三个：

- vma_create--创建 vma
- insert_vma_struct--插入一个 vma
- find_vma--查询 vma。

vma_create函数根据输入参数vm_start、vm_end、vm_flags来创建并初始化描述一个虚拟内存空间的vma_struct结构变量。insert_vma_struct函数完成把一个vma变量按照其空间位置[vma->vm_start,vma->vm_end]从小到大的顺序插入到所属的mm变量中的mmap_list双向链表中。find_vma根据输入参数addr和mm变量，查找在mm变量中的mmap_list双向链表中某个vma包含此addr，即vma->vm_start<= addr <vma->end。这三个函数与后续讲到的page fault异常处理有紧密联系。

涉及mm_struct的操作函数比较简单，只有mm_create和mm_destroy两个函数，从字面意思就可以看出是是完成mm_struct结构的变量创建和删除。在mm_create中用kmalloc分配了一块空间，所以在mm_destroy中也要对应进行释放。在ucore运行过程中，会产生描述虚拟内存空间的vma_struct结构，所以在mm_destroy中也要进对这些mmap_list中的vma进行释放。

4 Page Fault异常处理

对于第三节提到的第二个关键问题，解决的关键是page fault异常处理过程中主要涉及的函数 -- do_pgfault。在程序的执行过程中由于某种原因（页框不存在/写只读页等）而使 CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射时，CPU 会产生一次缺页异常，从而需要进行相应的缺页异常服务例程。这个缺页异常处理的时机就是求调页/页换入换出/处理的执行时机，当相关处理完成后，缺页异常服务例程会返回到产生异常的指令处重新执行，使得软件可以继续正常运行下去。

具体而言，当启动分页机制以后，如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误（比如写一个只读页或用户态程序访问内核态的数据等），就会发生缺页异常。产生页面异常的原因主要有：

- 目标页面不存在（页表项全为0，即该线性地址与物理地址尚未建立映射或者已经撤销);
- 相应的物理页面不在内存中（页表项非空，但Present标志位=0，比如在swap分区或磁盘文件上)，这将在下面介绍换页机制实现时进一步讲解如何处理;
- 访问权限不符合(此时页表项P标志=1，比如企图写只读页面).

当出现上面情况之一，那么就会产生页面page fault（#PF）异常。产生异常的线性地址存储在CR2中，并且将是page fault的产生类型保存在 error code 中，比如 bit 0 表示是否 PTE_P为0，bit 1 表示是否 write 操作。

产生缺页异常后，CPU硬件和软件都会做一些事情来应对此事。首先缺页异常也是一种异常，所以针对一般异常的硬件处理操作是必须要做的，即CPU在当前内核栈保存当前被打断的程序现场，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode；由于缺页异常的中断号是0xE， CPU把中断0xE服务例程的地址（vectors.S中的标号vector14处）加载到cs和eip寄存器中，开始执行中断服务例程。这时ucore开始处理异常中断，首先需要保存硬件没有保存的寄存器。在vectors.S中的标号vector14处先把中断号压入内核栈，然后再在trapentry.S中的标号__alltraps处把ds、es和其他通用寄存器都压栈。自此，被打断的程序现场被保存在内核栈中。接下来，在trap.c的trap

函数开始了中断服务例程的处理流程，大致调用关系为：

trap--> trap_dispatch-->pgfault_handler-->do_pgfault

下面需要具体分析一下do_pgfault函数。CPU把引起缺页异常的虚拟地址装到寄存器CR2中，并给出了出错码（tf->tf_err），指示引起缺页异常的存储器访问的类型。而中断服务例程会调用缺页异常处理函数do_pgfault进行具体处理。缺页异常处理是实现按需分页、swap in/out的关键之处。

ucore中do_pgfault函数是完成缺页异常处理的主要函数，它根据从CPU的控制寄存器CR2中获取的缺页异常的虚拟地址以及根据 error code的错误类型来查找此虚拟地址是否在某个VMA的地址范围内以及是否满足正确的读写权限，如果在此范围内并且权限也正确，这认为这是一次合法访问，但没有建立虚实对应关系。所以需要分配一个空闲的内存页，并修改页表完成虚地址到物理地址的映射，刷新TLB，然后调用iret中断，返回到产生缺页异常的指令处重新执行此指令。如果该虚地址不再某VMA范围内，这认为是一次非法访问。

4 页面置换机制的实现

4.1 页替换算法

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，会产生缺页异常。这时，操作系统必须能够应对这种缺页异常，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缺页异常次数少，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：

- 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），即在增加放置页的页帧的情况下，反而使缺页异常次数增多。
- 时钟（Clock）页替换算法：也称最近未使用 (Not Used Recently, NUR) 页替换算法。虽然二次机会算法是一个较合理的算法，但它经常需要在链表中移动页面，这样做既降低了效率，又是不必要的。一个更好的办法是把各个页面组织成环形链表的形式，类似于一个钟

的表面。然后把一个指针指向最古老的那个页面，或者说，最先进来的那个页面。时钟算法和第二次机会算法的功能是完全一样的，只是在具体实现上有所不同。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”。然后将内存中所有的页都通过指针链接起来并形成一个循环队列。初始时，设置一个当前指针指向某页（比如最古老的那个页面）。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，把它换出到硬盘上；如果访问位为“1”，这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。但该算法需要硬件支持来设置访问位，且该算法在本质上与FIFO算法是类似的，惟一不同的是在clock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。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。

4.2 页面置换机制

如果要实现页面置换机制，只考虑页替换算法的设计与实现是远远不够的，还需考虑其他问题：

- 哪些页可以被换出？
- 一个虚拟的页如何与硬盘上的扇区建立对应关系？
- 何时进行换入和换出操作？
- 如何设计数据结构已支持页替换算法？
- 如何完成页的换入换出操作？

这些问题在下面会逐一进行分析。注意，在实验三中仅实现了简单的页面置换机制，但现在还没有涉及实验四才实现的用户进程（激活页面置换的用户方）和内核线程（完成页面置换的服务方），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。

1. 可以被换出的页

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。

2. 虚存中的页与硬盘上的扇区之间的映射关系

如果一个页被置换到了硬盘上，那操作系统如何能简捷来表示这种情况呢？在ucore的设计上，充分利用了页表中的PTE来表示这种情况：当一个 PTE 用来描述一般意义上的物理页时，显然它应该维护各种权限和映射关系，以及应该有 PTE_P 标记；但当它用来描述一个被置换出去的物理页时，它被用来维护该物理页与 swap 磁盘上扇区的映射关系，并且该 PTE 不应该由 MMU 将它解释成物理页映射(即没有 PTE_P 标记)，与此同时对应的权限则交由 mm_struct 来维护，当对位于该页的内存地址进行访问的时候，必然导致 page fault，然后ucore能够根据 PTE 描述的 swap 项将相应的物理页重新建立起来，并根据虚存所描述的权限重新设置好 PTE 使得内存访问能够继续正常进行。

如果一个页（4KB/页）被置换到了硬盘某8个扇区（0.5KB/扇区），该 PTE的最低位--present位应该为0 （即 PTE_P 标记为空，表示虚实地址映射关系不存在），接下来的7位暂时保留，可以用作各种扩展；而原来用来表示页帧号的高24位地址，恰好可以用来表示此页在硬盘上的起始扇区的位置（其从第几个扇区开始）。为了在页表项中区别 0 和 swap 分区的映射，将 swap 分区的一个 page 空出来不用，也就是说一个高24位不为0，而最低位为0的PTE表示了一个放在硬盘上的页的起始扇区号（见swap.h中对swap_entry_t的描述）：

```
swap_entry_t
 ---------------------------------------------
 |        offset        | reserved  | 0 |
 ---------------------------------------------
          24 bits            7 bits    1 bit
```

考虑到硬盘的最小访问单位是一个扇区，而一个扇区的大小为512（2^8）字节，所以需要8个连续扇区才能放置一个4KB的页。在ucore中，用了第二个IDE硬盘来保存被换出的扇区，根据实验三的输出信息

“ide 1: 262144(sectors), 'QEMU HARDDISK'.”

我们可以知道实验三可以保存262144/8=32768个页，即128MB的内存空间。swap 分区的大小是 swapfs_init 里面根据磁盘驱动的接口计算出来的，目前 ucore 里面要求 swap 磁盘至少包含 1000 个 page，并且至多能使用 1<<24 个page。

3. 执行换入换出的时机

在实验三中， check_mm_struct变量这个数据结构表示了目前 ucore认为合法的所有虚拟内存空间集合，而mm中的每个vma表示了一段地址连续的合法虚拟空间。当ucore或应用程序访问地址所在的页不在内存时，就会产生 page fault异常，引起调用do_pgfault函数，此函数会判断产生访问异常的地址属于check_mm_struct某个vma表示的合法虚拟地址空间，且保存在硬盘swap文件中（即对应的PTE的高24位不为0，而最低位为0），则是执行页换入的时机，将调用swap_in函数完成页面

换入。

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，只是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。

在实验三中的基本练习中，支持上述第二种种情况。对于第一种积极换出策略，即每隔1秒执行一次的实现积极的换出策略，可考虑在扩展练习中实现。对于第二种消极的换出策略，则是在ucore调用alloc_pages函数获取空闲页时，此函数如果发现无法从物理内存页分配器（比如first fit）获得空闲页，就会进一步调用swap_out函数换出某页，实现一种消极的换出策略。

4. 页替换算法的数据结构设计

到实验二为止，我们知道目前表示内存中物理页使用情况的变量是基于数据结构Page的全局变量pages数组，pages的每一项表示了计算机系统中一个物理页的使用情况。为了表示物理页可被换出或已被换出的情况，可对Page数据结构进行扩展：

```
struct Page {
    ……
    list_entry_t    pra_page_link;
    uintptr_t       pra_vaddr;
    };
```

pra_page_link 可用来构造按页的第一次访问时间进行排序的一个链表，这个链表的开始表示第一次访问时间最近的页，链表结尾表示第一次访问时间最远的页。当然链表头可以就可设置为 pra_list_head（定义在 swap_fifo.c 中），构造的时机实在 page fault 发生后，进行 do_pgfault 函数时。pra_vaddr 可以用来记录此物理页对应的虚拟页起始地址。

当一个物理页 （struct Page） 需要被 swap 出去的时候，首先需要确保它已经分配了一个位于磁盘上的swap page（由连续的8个扇区组成）。这里为了简化设计，在swap_check函数中建立了每个虚拟页唯一对应的swap page，其对应关系设定为：虚拟页对应的PTE的索引值 = swap page的扇区起始位置*8。

为了实现各种页替换算法，我们设计了一个页替换算法的类框架swap_manager:

```
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);
};
```

这里关键的两个函数指针是map_swappable和swap_out_vistim，前一个函数用于记录页访问情况相关属性，后一个函数用于挑选需要换出的页。显然第二个函数依赖与第一个函数记录的页访问情况。tick_event函数指针也很重要，结合定时产生的中断，可以实现一种积极的换页策略。

5. swap_check 的检查实现

下面具体讲述一下实验三中实现置换算法的页面置换的检查执行逻辑，便于大家实现练习2。实验三页面置换的检查过程在函数swap_check （kern/mm/swap.c中）中，其大致流程如下。

1. 调用mm_create建立mm变量，并调用vma_create创建vma变量，设置合法的访问范围为4KB~24KB；
2. 调用free_page等操作，模拟形成一个只有4个空闲 physical page；并设置了从4KB~24KB的连续5个虚拟页的访问操作；
3. 设置记录缺页次数的变量pgfault_num=0，执行check_content_set函数，使得起始地址分别对起始地址为0x1000, 0x2000, 0x3000, 0x4000的虚拟页按时间顺序先后写操作访问，由于之前没有建立页表，所以会产生page fault异常，如果完成练习1，则这些从4KB~20KB的4虚拟页会与ucore保存的4个物理页帧建立映射关系；
4. 然后对虚页对应的新产生的页表项进行合法性检查；
5. 然后进入测试页替换算法的主体，执行函数check_content_access，并进一步调用到_fifo_check_swap函数，如果通过了所有的assert。这进一步表示FIFO页替换算法基本正确实现；
6. 最后恢复ucore环境。

5 实验报告要求

从网站上下载lab3-2012.zip后，解压得到本文档和代码目录 lab3-2012，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab3-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有“LAB3”的注释，代码中所有需要完成的地方都有“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。所有扩展实验的加分总和不超过10分。

附录：正确输出的参考：

```
yuchen@yuchen-PAI4:~/oscourse/2012spring/lab3/lab3-code-2012$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc010962b (phys)
  edata  0xc0122ac8 (phys)
  end    0xc0123c10 (phys)
Kernel executable memory footprint: 143KB
memory management: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
```

```
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 1, total 31992
 mm->sm_priv c0123c04 in fifo_init_mm
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!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
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
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
check_swap() succeeded!
++ setup timer interrupts
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:20:
    EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```