Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

孙绍聪 2000012977@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

1.从这次开始我准备用中文写文档 😃

2.关于新增的文件：

- vm/:page.c page.h frame.c frame.h swap.c swap.h

  这些文件将在后续说明。

- filesys/:filesys_lock.h

  为了方便编码，我将lab2中定义的文件系统锁单独写在了一个头文件中，不再后面文档中赘述。

3.本机测试结果

tests/vm/Rubric.functionality 55/ 55 50.0%/ 50.0%
tests/vm/Rubric.robustness 28/ 28 15.0%/ 15.0%
tests/userprog/Rubric.functionality 108/108 10.0%/ 10.0%
tests/userprog/Rubric.robustness 88/ 88 5.0%/ 5.0%
tests/filesys/base/Rubric 30/ 30 20.0%/ 20.0%
Total 100.0%/100.0%

ps. page.h page.c frame.c这三个文件的开头处各有一段大注释，可能对阅读我的文档有帮助。

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

关于demand page的部分设计和我的室友谭亦轩讨论过，所以我们可能有部分设计比较相似。

Page Table Management

DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

page.h中对未呈现的页(unpresent page)在页表中PTE的结构进行定义：

```
/* Some basic macros to get each field. */
#define SPTE_ALIGN 8                    /**< Alignment. */
#define SPTE_MASK ~(SPTE_ALIGN-1)       /**< SPTE Pointer. */
```

```
#define SPTE_W 0x4                      /**< Writable flag. */
#define SPTE_S 0x2                      /**< Swap flag. */
```

page.c中定义了SPTE的结构及辅助数据结构：

```
union spte
{
    struct list_elem elem;              /**< List elem in empty_spte. */
    struct{
        struct file *file;              /**< Which file to load the page. */
        size_t position;                /**< Position in file. */
        struct semaphore ready_sema;    /**< Synch loading and evicting. */
    }content;
};

/* To make sure pointers to SPTEs are aligned. Its size must be rounded up to
multiple of SPTE_ALIGN = 8.*/
#define SPTE_SIZE ROUND_UP(sizeof(union spte),SPTE_ALIGN)

static struct list empty_spte;          /**< List for empty STPEs. */
static struct lock empty_spte_lock;     /**< Lock to protect empty_spte. */
```

ALGORITHMS

> A2: In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

这里主要描述**推迟加载**(lazy loading)的过程： 采用未呈现的页的PTE作为对SPT元素(SPTE)的索引，在SPTE中保存文件的存储位置，并利用了PTE中空余的位指明对SPTE存储信息的解释方式(解释为文件系统的位置还是交换文件中的位置，以下用"解释位"指代)及该用户页面的可读写情况(以下用"读写位"指代)。建议参考page.h和page.c上方的大注释。

- **创建SPTE**：对于每一个需要加载的虚拟页面(这个页面可能对应于可执行文件中的段、用户栈的页面或是来自交换文件)，从empty_spte链表中取一个空的SPTE，在其中填上页面加载的文件位置，然后将这个SPTE的指针及这个虚拟页面的解释位和读写位打包成PTE存储在页表中
- **使用SPTE进行加载**：当用户进程产生了一个page fault时，如果这是一个未呈现的页面，page fault handler会尝试通过调用pagedir_demand_page()来呈现这个页面。首先解读PTE获取SPTE指针及解释位和读写位信息并通过frame_get_page()请求一个物理页面，然后按照解释位指明的方式解读SPTE中存储的文件位置信息，从对应的文件位置读取数据填充在之前请求的物理页面上，最后改写用户虚拟页面的PTE和物理页面对应的FTE来声明物理页面被这个用户进程获取。
- **释放一个SPTE**：用户程序结束的时候，其页表中可能有PTE存储着SPTE的信息，这些SPTE仍然被这个用户进程所占有，需要被释放。所以用户进程调用pagedir_destory()时会检查每个PTE是否是SPTE的索引，如果是则调用free_spte()来释放，释放操作是将这个SPTE插入empty_spte链表以供其他进程使用，释放一个存储的文件位置在交换文件中的SPTE还需要释放交换文件中的槽(swap slot)。

> A3: How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?

- 不论是用户的主动访问还是进行系统调用时内核的访问，我都会使用用户虚拟地址进行。内核虚拟地址只在加载用户虚拟系页面和驱逐一个页面的时候会被使用，但是这些都不是用户的主动访问，无需考虑。所以我不需要考虑内核的access bit和dirty bit。

SYNCHRONIZATION

> A4: When two user processes both need a new frame at the same time, how are races avoided?

- 在pagedir_demand_page()处理page fault时使用了一把锁pagedir_lock，在取得SPTE之前获取这把锁，在调用frame_get_page()请求一个可用的物理页面之后释放这把锁，所以两个进程对frame_get_page()的调用是互斥的，不会产生冲突。

- ps. 如果仅仅考虑请求物理页面的这一过程的互斥，似乎并不需要这把锁来解决，因为这把锁的真正目的是解决P驱逐Q的页面时对Q的页表条目的修改和Q自身对该页表条目的访问产生的冲突而设置的，使得两个进程对frame_get_page()的调用变得互斥是一个副作用。即使不考虑这把锁的作用，两个进程可以并发的调用frame_get_page()。该函数内部调用的palloc_get_page()和select_frame_to_evict()分别受到userpool的锁和frame_list的锁的保护，都可以看作是原子的。另外，物理页面和FTE是一一绑定的，获取物理页面的同时会固定获取其对应的FTE，反之亦然。以上两条保证了竞争不会发生。

- pps. 这个问题感觉放在Part B来问更合适.....

RATIONALE

> A5: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

- 另一种组织SPT的方式是建立一个和PT结构相似的数据结构，采用相同的组织方式，只不过将PTE换做SPTE的索引。但是一个明显的问题是对空间的浪费，实现上也并不比我采用的方法简单。PTE的1~31位在P=0时是未被使用的，但是我没有想出一种能够将文件位置的完整信息(包括处于文件系统还是交换文件、具体的文件或槽、文件中的具体位置和读取的字节数等)全部集中到这31位上的方法。于是我采用了一个折中的方法：将PTE高位的部分解释成一个指针，指向一个包含文件位置信息的外部的结构SPTE，将两个低位用于保存两个Flag信息。
- 这种SPTE结构是可以在不同的进程之间"流通"的，每个进程可以从全局SPTE池(empty_spte链表)中自取所需，全局SPTE池会保障供给充足，进程使用完再归还全局SPTE池。这种管理SPTE的方式还带来了一个优点，管理SPTE相关的结构(list_elem)不会再SPTE有效的时候发挥作用，正因如此，我将SPTE声明成了一个联合(union)的形式，处于链表中的空SPTE按照链表元素的方式来解释这个联合，被安插进SPT的SPTE按照它应包含的内容的形式来解释这个联合。

Paging To And From Disk

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

frame.c中定义了frame_table及其中元素FTE(建议参考frame.c上方的大注释)：

```
struct fte{
    struct list_elem elem;              /**< List element in frame_list. */
```

```
    uint32_t *pd;                       /**< Mapped pagedir. */
    void *upage;                        /**< Mapped user virtual page. */
    struct file* file;                  /**< Loaded from which file. */
    size_t position;                    /**< Position in file. */
    void *evict_spte;                   /**< Synch loading and evicting. */
};

static struct list frame_list;          /**< List for all active FTEs. */
static struct lock frame_list_lock;     /**< Lock to protect frame_list. */
static struct fte *frame_table;         /**< Array contains all FTEs. */
static struct list_elem *clock_ptr;     /**< For clock algorithm. */
```

swap.c中定义了swap file的组织结构：

```
/** Number of sectors in a slot. */
#define SECTOR_PER_SLOT (PGSIZE/BLOCK_SECTOR_SIZE)

static struct block *swap_block;        /**< Block device for swap. */
static struct bitmap *swap_slots;       /**< Manage all slots. */
static struct lock swap_slots_lock;     /**< Lock to protect swap_slots. */
```

ALGORITHMS

> B2: When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

- 我采用了clock algorithm来选择一个物理页面进行驱逐，clock_ptr会循环遍历frame_list(链表中的FTE都是属于已分配的页面)，检查这个页面的对应虚拟页面的PTE中的access bit(A位),如果发现A位是1则将其置为0继续遍历，如果第一次找到某一个物理页面的A位是0则选择这个物理页面驱逐。由于A3中的解释，我不需要考虑这个物理页面对应的内核虚拟页面的PTE。

> B3: When a process P obtains a frame that was previously used by a process Q, how do you adjust the page table (and any other data structures) to reflect the frame Q no longer has?

- 当P选择了Q的一个物理页面进行驱逐时，他会将这个物理页的FTE移除frame_list，根据FTE中存储的文件位置信息和这个页面的修改情况(获取Q页表项PTE中的dirty bit)决定将这个页面直接丢弃还是写入交换文件(lab3b中新增了一种写入文件系统的情况),根据这个决定产生相应的SPTE和PTE，用PTE覆盖Q的页表中对这个物理页的用户引用，最后还要将做出的决定暂时记录在和物理页绑定的FTE中。此后Q对这个页面的访问会产生page fault。

SYNCHRONIZATION

> B5: Explain the basics of your VM synchronization design. In particular, explain how it prevents deadlock. (Refer to the textbook for an explanation of the necessary conditions for deadlock.)

```
lab3a新增了的同步变量：
 - 保护empty_spte链表的empty_spte_lock
 - 保护frame_list链表的frame_list_lock
 - 保护swap_slots bitmap的swap_slots_lock
 - 用于同步进程之间访问对方页表和进程各自访问自身页表的 pagedir_lock
 - 每个SPTE中用于同步页面驱逐和页面加载的ready_sema

用到的之前定义过的同步变量：
 - 保护文件系统的filesys_lock
```

- lab3a新增的前三把锁均属于获取之后执行一段不会等待的关键路径，执行完成马上释放的类型，不满足Hold and wait的条件，不会产生死锁
- ready_sema主要用于进程间通信，也不会导致死锁
- 集中讨论pagedir_lock和filesys_lock之间的交互：pagedir_lock获取之后有可能获取filesys_lock，但是所有的首先持有filesys_lock的过程(即有关文件操作的系统调用)中都不会产生page_fault，也就没有对pagedir_lock的需求，所以这两把锁的交互不会产生死锁。有关如何在存在文件操作的系统调用中避免产生page fault：在系统调用获取到filesys_lock之前，先将用户地址处的字串复制到内核中新开辟的内存，稍后获取filesys_lock使用内核中的内存区域地址进行参数传递，处理完系统调用释放filesys_lock之后再释放这段内存。

> B6: A page fault in process P can cause another process Q's frame to be evicted. How do you ensure that Q cannot access or modify the page during the eviction process? How do you avoid a race between P evicting Q's frame and Q faulting the page back in?

- 有关如何避免Q对正在被驱逐的页的访问：就像B3中讨论的那样，决定驱逐后立刻修改Q的页表项，Q之后的访问会产生page fault，page fault之后的操作是下一段要回答问题。这里还有一个细微的问题，Q在P决定驱逐这个页面之后且P修改Q的页表项之前进行访问，这种访问可以正常进行并且会得到正确的结果，因为P并没有修改这一页上存储的内容。
- 有关同步P的驱逐过程和Q的加载过程：首先pagedir_lock保证了Q会获取到正确的SPTE。其次使用了B5中提到的ready_sema。如果P判定这个驱逐过程需要进行I/O操作，会保持Q加载时需要用到的SPTE中的ready_sema处于一个未准备好的状态(即value为0)，直到P稍后完成了I/O(即完成了驱逐过程)，P会调用ready_spte()使得SPTE处于准备好的状态。Q只有在SPTE处于准备好的状态时才可以进行加载操作(或释放操作)。

> B7: Suppose a page fault in process P causes a page to be read from the file system or swap. How do you ensure that a second process Q cannot interfere by e.g. attempting to evict the frame while it is still being read in?

- 只有位于frame_list中的页面才是可能被驱逐的候选，而一个页面只有在完成了加载过程后才会被插入frame_list，所以不会驱逐正在被加载的页面。

> B8: Explain how you handle access to paged-out pages that occur during system calls. Do you use page faults to bring in pages (as in user programs), or do you have a mechanism for "locking" frames into physical memory, or do you use some other design? How do you gracefully handle attempted accesses to invalid virtual addresses?

- 有关如何在系统调用中避免死锁已经在B5中讨论过了。

- 我采用了和处理用户态的page fault相同的机制来处理。系统调用过程中可能发生的page fault都存在于准备调用参数的过程中，包括获取系统调用号，获取各个参数和复制用户的字符串，唯一的例外时read系统调用会写入用户页面。在这些位置发生page fault分两种类型。第一种是页面已呈现，但是非法访问的类型，系统会像lab2的处理一样返回一个错误代码到系统调用处理程序，由系统调用处理程序结束进程。第二种是页面未呈现，系统会首先尝试进行页面加载，如果加载失败，则同样像lab2的处理返回错误代码到系统调用处理程序，如果加载成功，则会重复一遍导致page fault的指令，指令执行的结果只能是触发第一种类型的page fault或执行成功。(不考虑加载页面后重复执行指令之前页面又被换出的情况，这种情况只会重复触发第二种类型的page fault导致再次加载，而且是小概率事件。)

RATIONALE

> B9: A single lock for the whole VM system would make synchronization easy, but limit parallelism. On the other hand, using many locks complicates synchronization and raises the possibility for deadlock but allows for high parallelism. Explain where your design falls along this continuum and why you chose to design it this way.

- 我的实现可能是介于第一种(一把大锁)和第二种(许多小锁)之间，更偏向第二种。
- 至于选择的原因：首先，当然是因为这样能允许表现还不错的并行度，虽然对于Pintos的性能(至少对测试点)影响可能并不是非常大。其次，采用一把大锁的设计并不利于将驱逐的页面写入文件和从文件中加载页面这两个过程从锁中独立出来，也不利于对于程序语义的清晰表达，因为VM中存在多种不同需求的同步，只用一把锁语义会不明确。最后，其实我也考虑过将pagedir_lock拆分成许多的小锁存放在各个进程的结构中，但是这样一是增加实现难度，二是并不会带来很多性能上的提升(因为pagedir_lock保护的关键路径是执行时间比较短的路径，不涉及I/O)，所以我最终放弃了这种想法。