Project 3b: 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.

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%

> 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.

无

Stack Growth

ALGORITHMS

> A1: Explain your heuristic for deciding whether a page fault for an invalid virtual address should cause the stack to be extended into the page that faulted.

- 有关合法栈增长的判断：
  - 产生page fault的地址落在**用户栈指针esp-32字节**到**用户栈栈底，即PHYS_BASE**的范围内。
  - 规定了用户栈的最大大小为8M，即要求在满足上一条的同时，产生page fault的地址也必须大于等于**PHYS_BASE-8MB。**
- 有关是否需要为用户栈添加一个新的映射(是否进行栈增长)：
  - 这个产生page fault的地址必须是一个合法的栈增长。
  - 这一地址所处的虚拟页之前未被映射过(PTE=0)。

ps. 这里关于合法栈增长的判断会将用户直接访问esp以下32字节范围的空间视为合法(通过PUSH和PUSHA以外的指令，如MOV)。严格上讲这是不合理的，因为这种访问是非法的。更加严谨的判断可以根据产生page fault的指令的不同来界定允许访问的下界，但是据助教的回答，这种判断没有必要。

Memory Mapped Files

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.

thread.h中的新定义：

```
/** Thread's mmap definition. */
typedef int mmapid_t;                       /**< Mmapid. */
#define INIT_MMAPID ((mmapid_t)0)           /**< Lowest mmapid. */
#define MMAPID_ERROR ((mmapid_t)-1)         /**< Error mmapid. */
/** A mapping struct. */
struct mmap_t
{
   struct list_elem elem;                   /**< List elem in mmap_list. */
   mmapid_t mmapid;                         /**< Mmapid. */
   struct file *f;                          /**< Mapping file. */
   void *addr;                              /**< Mapping start address. */
   off_t length;                            /**< Mapping total length. */
};
```

thread struct中的新成员：

```
struct list mmap_list;              /**< A list of its all mappings. */
mmapid_t mmapid_allocater;          /**< Generate unique mmapid in thread. */
void *user_esp;                     /**< Saved user stack pointer. */
```

ALGORITHMS

> B2: Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages.

- 映射了文件的虚拟页和其它类型的虚拟页并没有本质区别，都遵循从spte获取加载信息，驱逐或释放时参考FT的过程。所以内存映射文件仅仅是做spte的设置，而当页面被驱逐或释放时注意将修改的页写回文件即可。
- page fault触发时，如果spte上的信息表明这个页来自交换分区，则从交换分区读取页面并释放交换槽；否则从文件系统读取页面。页面被驱逐时，只有被修改且无法写回的页面会被驱逐到交换分区，其他页面会被驱逐回文件系统。

> B3: Explain how you determine whether a new file mapping overlaps any existing segment.

- 在这个进程的页表中进行检查，新的映射不与其他段重叠当且仅当页表中新的映射会覆盖的页的页表项(PTE)均为未映射的状态(全零)。

RATIONALE

> B4: Mappings created with "mmap" have similar semantics to those of data demand-paged from executables, except that "mmap" mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain why your implementation either does or does not share much of the code for the two situations.

- 这两种不同的内存映射方式在触发page fault做demand page时，物理内存不足做evict page时和进程退出释放物理内存时都遵循相似的方式，区别是读取和写回的目的地不同。mmap的文件永远是从文件系统中读入，根据页面是否被修改决定是否写回文件系统。映射可执行文件的页面的第一次读入会从文件系统中读入，之后的读入可能从文件系统读入也可能从交换文件读入，根据页面的修改情况决定是否写回文件系统。区分二者的一个关键特点是**被映射的文件是否可写**，这一点可以通过新增的接口file_can_write()判断。
- 在我的实现中，二者使用相同的demand page过程。evict page过程采用了相同的接口，当某一页将要被驱逐时，通过判断文件是否可写来决定被修改过的页将会被写回文件系统还是将会被写入交换文件。唯一不共用的过程是释放物理页的时候，mmap映射的页可能会在程序运行过程中被unmap释放，也可能在进程退出时自动释放，这两种释放都是通过pagedir_unmap_page()实现的，通过这个函数释放物理页会考虑写回文件。映射可执行文件的页是通过thread_exit()调用的pagedir_destory()释放的。由于这个函数被调用的时候可执行文件已经被关闭，对可执行文件进行的file_can_write()检查会返回不可预测的结果，所以pagedir_destory()释放物理页是不会尝试写回文件(调用这个函数时页表里仍存在的映射只有可执行文件和用户栈，他们也都不需要写回)。