```
          +---------------------------+
          |          CS 130           |
          | PROJECT 3: VIRTUAL MEMORY |
          |      DESIGN DOCUMENT      |
          +---------------------------+
```

---- GROUP ----

>> Fill in the names and email addresses of your group members.
Group 08

Yi Qian <qianyi1@shanghaitech.edu.cn>
Guancheng Li <ligch@shanghaitech.edu.cn>

---- PRELIMINARIES ----

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

>> 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.

CSCI 350: Pintos Guide

https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

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.

In "threads/thread.h"

```
struct thread
{
   ...
   struct hash* page_table;           /* Supplemental page table */
}
```

In "vm/page.h"

```
/* Three different page type. */
#define PAGE_FROM_FILE 0
#define PAGE_FROM_SWAP 1
#define PAGE_FROM_SEGM 2

struct sup_page_table_entry
{
   void *uaddr;                        /* user page's virtual address. */
   struct thread *owner_thread;        /* The entry's owner thread. */
   struct frame_table_entry *frame_table_entry;/* The page's corresponding
```

```
frame entry. */
      struct hash_elem elem;       /* Hash element. */

      bool writable;               /* Whether the page is writable. */
      bool using;                  /* Whether the page is using. */
      int page_type;               /* 0 for file, 1 for swap, 2 for segment. */

      struct file *file;           /* The page's mapped file. */
      off_t offset;                /* The starting offset in the file. */
      uint32_t read_bytes;         /* How many bytes are read. */
      uint32_t zero_bytes;         /* How many zero bytes at the end. */
      int block_start_index;       /* Start index in swap disk. */
      struct list_elem l_elem;     /* List element. */
   };

---- ALGORITHMS ----

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

   Create an SPT entry and assign the given page's user virtual address to it.
   Go through the hash table of SPT, and if the page is valid, an SPT entry is
returned.
   All data can be accessed through the returned SPT entry.

>> 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?

   We always access data through user virtual address, so only user pages will
be dirty.
   Dirty pages are handled when the frame is swapped out.

---- SYNCHRONIZATION ----

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

   The frame_table has a lock called frame_table_lock, which only allows one
   process change the frame_table simultaneously.

---- RATIONALE ----

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

   Because we need to keep more data for pages and frames as we are performing
   more functionalities like swap, mmp, etc.

             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.
```

```
   In "vm/frame.h"
   struct frame_table_entry
   {
      struct list_elem elem;             /* element for frame table */
      struct lock lock;                  /* lock of one frame */
      uint8_t *frame;                    /* point to the data of the frame */
      struct sup_page_table_entry *page; /* corresponding page */
   };

   struct list frame_table;                /* Store all the frame allocated. */
   struct lock frame_table_lock;           /* Lock of frame table. */

   In "vm/swap.h"
   #define BLOCKS_PER_PAGE 8
   static struct block* global_swap_block; /* Block used to swap. */
   struct lock blocks_lock;                /* Lock of blocks bitmap. */
   struct bitmap* blocks_bitmap;           /* Records of which block are free.
*/

---- 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.

   We use the clock policy.
   The clock hand is initialized to point to the first frame in list
   frame_table.
   If the pointed frame's corresponding page's isn't using, we evict it.
   Otherwise, clock hand should point to the next frame. If it point to the end
   of frame_table, we adjust it to point to the begin of frame_table.
   Do this recursively until we find a frame which corresponding page isn't
   using.

>> 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?

   We should decide the frame's corresponding page type first. If it's
   PAGE_FROM_FILE, we write it to the file using function
   write_page_to_file(page), if it's PAGE_FROM_SWAP,
   we swap it out. Then use pagedir_clear_page(Q->pagedir, page->uaddr) to
   clear the page from Q's page directory. Finally, let the frame's
   corresponding page be NULL.
   The frame can be allocated to P.


>> B4: 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.

   When page fault occurs, we firstly use pg_round_down(faddr) to get the
   nearest page address and search in supplemental page table using this
   address. If we can't find a page, it means that it's possible for us to
   extend the stack.
   To ensure this, we should do some checks. First, the new page address should
   less than PHYS_BASE - STACK_MAX(larger than stack max capacity). Second, the
```

fault address should less than esp - 32.

Only when the conditions mentioned above are all satisfied, we will extend the stack.

---- 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.)

Each sup_page_table_entry has a boolean value to indicates whether the page is being used.

If it's true, we can't evict it.

We use a frame_table_lock to ensure frame_table can be accessed only by one process simultaneously.

We add a lock in each frame to ensure that only one process can change certain frame at the same time.

We use blocks_lock to ensure that only one process can change blocks_bitmap at the same time.

>> 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?

Each frame has its own lock. So we can ensure only one process can change the frame state simultaneously. Besides, if one frame's corresponding page is being used, we also cannot evict the frame. We we are evicting the frame, we will set its corresponding page's boolean using to true to prevents other process accessing the page.

>> 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?

Each frame has its own lock. So we can ensure only one process can change the frame state simultaneously. If P will read from the file system or swap to one frame, it should acquire the frame's own lock first. It ensure that when P is reading or swapping, Q can't change the frame's state.

>> 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?

We first check whether the fault address is valid, if it isn't, the process will exit(-1) directly.

If the fault address is valid but not in user mode, we load the page manually instead of using the same handler as in user programs.

Each frame has its own lock and each sup_page_table_entry has it own state boolean using to indicates whether the page is being used.

---- 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.

We use a lot locks to ensure the high parallelism.
A lock for operations of files: lock_of_filesys
Each frame has its own lock.
A lock for frame_table: frame_table_lock.
A lock for swap's bitmap: block_lock

Each lock has its own function, it make the whole structure clearer.

MEMORY MAPPED FILES
==========

---- DATA STRUCTURES ----

>> C1: 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.

In "threads/thread.h"
typedef int mapid_t;

```
struct mmap_info
{
   mapid_t mapid;              /* Record the return value of mmap. */
   struct file* mapped_file;   /* The file mapped to memory. */
   struct list mmap_pages;     /* Store the related sup_page_table_entry. */
   struct list_elem elem;      /* List element. */
};
struct thread
{
   ....
   mapid_t mmap_num;         /* Which number should be allocated to mmap. */
   struct list mmap_infos;   /* Store the processes mmap info. */
   ...
};
```

---- ALGORITHMS ----

>> C2: 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.

The page type PAGE_FROM_FILE is designed for the memory mapped files.
We should construct a mmap_info each time calling mmap function. The mapid in mmap_info if allocated by mmap_num stored in process, and mapped file stores the file mapped to memory. The mmap_info should be stored in rocess's list mmap_infos.
When we mapped a file, we should reopen the file first and then store the file's information in sup_page_table_entry. The information is a pointer to the file, starting at which byte(offset), how many bytes read in the page(read_bytes) and how many zero bytes in the end of the page(zero_bytes).

Then we add these sup_page_table_entry into process's supplemental page table and mmap_info's list mmap_pages.

When page fault comes across, the page fault handler notices that the page type is PAGE_FROM_FILE, it load needed bytes from file according to the information stored in sup_page_table_entry.

When evict the page which page type is PAGE_FROM_FILE, the evict method is different from swap. It should write bytes in frame to the file according to the information stored in sup_page_table_entry instead of swap to the swap disk.

>> C3: Explain how you determine whether a new file mapping overlaps
>> any existing segment.

The arguments passed to the function contains the mapped starting address(addr).

And we know which file to be mapped, so we can get the file's size.

First, we should check the address passed to mmap function. Because it's address aligned, the addr % PGSIZE should equal to zero.

Then we should check address between addr and addr + file size. We should round down the address to get the page address and find the page according to page address in process's supplemental page table. If we can find, it indicates that there will be an overlap. So we should stop mmap. If all addresses between addr and addr + file size are checked without overlap, the mmap will return right mapid instead of MAPID_ERROR.

---- RATIONALE ----

>> C4: 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.

We haven't implemented the shared method because currently the implementation is designed for each process, which means each process has its own mmap infos.

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?