+--+
| CS 140 |
| PROJECT 3: VIRTUAL MEMORY |
| DESIGN DOCUMENT |
+--+

---- GROUP ----

Name	email
Xiaojing Hu	xhu7@buffalo.edu
Jingyi Zhao	zhao.elton@gmail.com
Zishan Liang	zsliang@buffalo.edu

---- PRELIMINARIES ----

Reference:

1. *Pintos Projects: Introduction*
   https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html
2. *OS Practice Project 3 and Design Document:*
   http://www.doc88.com/p-9746122546098.html
3. *CIS 520 project, Kansas Univerity:*
   https://github.com/ChristianJHughes/pintos-project3

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.

A:
For supplementary page table:

```
struct spt_entry {
   void *addr;                  /* User virtual address. */
   struct thread *thread;       /* Thread that owns current spt */
   struct frame *occupied_frame;       /* Page frame. */
   block_sector_t sector;       /* Starting sector of swap area, or -1. */
   bool read_only;              /* Read-only  */
   bool pinned;                 /* Cannot write back when false, if true, allow swap */
```

```
    struct file *file_ptr;              /* File. */
    off_t file_offset;                  /* Offset in file. */
    off_t file_bytes;                   /* Bytes to read/write, 1...PGSIZE. */
    struct hash_elem hash_elem;         /* Struct thread `pages' hash element. */
};
```

For frame table:

```
struct frame {
    struct lock lock;                   /* one access at a time */
    void *base;                         /* Kernel virtual base address. */
    struct spt_entry *pte;              /* Mapped process page, if any. */
    struct list_elem elem;
};
```

In struct thread we add:

```
struct list list_mmap_files;      /* Memory-mapped files. */
```

In thread.h we add:

```
struct mapping
{
    struct list_elem elem;          /* List element. */
    int map_handle;                 /* Mapping id. */
    struct file *file;              /* File*/
    uint8_t *base;                  /* memory mapping starts here : ) */
    size_t page_cnt;                /* Number of pages mapped. */
};
```

---- ALGORITHMS ----

>> **A2:** In a few paragraphs, describe your code for locating the frame,
>> if any, that contains the data of a given page.

A: We add the page struct in page.h, which has a member *frame pointing to the frame allocated to it. If page is not allocated a frame, the *frame member will point to NULL. And also we implement a struct frame which contains a pointer to the page occupying it. At the beginning, all *page in the frame struct point to NULL. We implement a function frame_allocate_and_lock() to allocate and lock a frame for a requesting page.
The frame_allocate_and_lock() function make a few attempts to get a new frame for the page. In the function we loop around the frame table to see if there is any frame that has not been occupied(its *page member points to NULL), if we find one, we allocate it to the page. If no frame is free, we then decide to evict a page and allocate the freed frame for the page.

>> **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?

A: In this implementation, we only access file through spt_entry or user address. We do not use kernel address.

---- SYNCHRONIZATION ----

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

A: We design a lock we make sure that at one time only one process can be allocated a frame.

---- RATIONALE ----

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

A: Once the page and the frame are mapped, we can find each other either in frame or in page, which make it very convenient for us to get the information we want.

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.

A: None

---- 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.

A: We use the LRU algorithm to evict the frame. The implementation is like CLOCK algorithm. We loop through frame table, and see whether the page it points to is accessed or not. If it has not been accessed, the we consider it as a least recently accessed page and evict it. If it has been accessed, we set it to not-accessed.

>> **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?

A: We first acquire the lock of the frame to avoid Q from accessing it. Then we check the dirty bit of the associated page and decide if we need to write the content to the disk(file) or do a swap. After that, we delete the page from Q's page table. When Q would like to access the frame, it will page faults.

>> **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.

A: There are two checks. First we check if the address is not beyond the maximum size of the stack. We think if the addr is not more than esp+32, there is a PUSH or PUSHA instruct. Then we allocate a new page for 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.)

A: Deadlock happens when the resource is not enough and there is circular dependency.
In our code, we eliminate every potencial circular dependency such that no deadlock happens.

>> **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?

A: We acquire the target frame's lock before evict it. In this way if Q wants to access the frame, it will be blocked.

>> **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?

A: When do swap in, we first acquire the frame's lock. And when we try to evict a frame, we need to acquire its lock too. The frame's lock makes sure that these two operation cannot happen together.

>> **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?

A: After the loading is finished, the system would go to page faults to bring in pages. If you cannot find the page in thread's SPT, then you are providing an invalid virtual address. At this case, we determine if the program is trying to access the stack. This can be done by checking if this address is within the bounds of max stack space, and if this address is accessing an address within 32 bytes. If both is satisfied, we allocate a new stack for this page(stack growth).

---- 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.

A: To make our design simple, we use as few locks as possible. There is a scan lock in the whole system to make sure the frame table can be traversed atomically. And a each frame has a lock to make sure it can only be accessed by one thread. These locks are acquired only when we access these data structures to improve the parallelism but still conserve our synchronization property.

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.

A:In syscall.c we add:

```
struct mapping {
   struct list_elem elem;       /* List element. */
   int handle;                  /* Mapping id. */
   struct file *file;           /* File. */
   uint8_t *base;               /* Start of memory mapping. */
   size_t page_cnt;             /* Number of pages mapped. */
};
```

---- 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.

A: Memory mapped files are encapsulated in the ‘mapping’ struct in thread.h’. It contains a mapping id, a mapping file, an address which is the start of the memory mapping. Each thread contains the list of files mapped to the thread, so as to copy the selected file directly to the memory.
The key is, that non-file related pages are moved to a swap partition upon eviction, regardless of whether or not the page is dirty. When evicted, the memory mapped file pages have to be written back after checking if it is modified. Otherwise, there will be no writing.

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

A: Only when a page is free and unmapped then the system will call pte_allocate to map a new file. Besides, it will iterate the existing file mappings to check if any space that is already occupied. If occupied, then refuse the allocating operation.

---- 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.

A: A page will be paged out ultimately. The key is that it has to decide whether the page should be written back out to the disk or not. If the page is marked by the flag, then it should be swapped, otherwise it should be written out to the file on disk.