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

---- GROUP ----

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

Zhaozheng Shen <shenzhzh@shanghaitech.edu.cn>
Wenhui Qiao   <qiaowh@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. (by Stephen Tsung-Han Sher)
A tutorial slides provided by Verginia Tech University

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.

Add a pointer to the supplemental page table in the struct of thread
as below:

```
struct thread{
        struct splmt_page_table *splmt_page_table;
}
```

The table mentioned above has a struct hash as the data structure to store
the entries, as defined below.

```
struct splmt_page_table{
        struct hash splmt_pages;
};
```

Here is the definition of the page table entry.

```
struct splmt_page_entry{
        struct hash_elem elem;     /* Used to be put in the hash. */
        enum splmt_page_type type; /* The type of the page. */
        uint8_t* user_vaddr;       /* The user virtual address of
the page. */
        uint8_t* frame_vaddr;      /* The physical address of the
page. */

        struct file* file;         /* A pointer to the file of this
page. */

        uint32_t valid_bytes;      /* Read bytes of the page, non-
zero. */
        uint32_t zero_bytes;       /* Zero bytes of the page,
always zero. */
        uint32_t offset;           /* The offset of this page. */
        unsigned swap_idx;         /* Indicate where it is stored
if swapped. */

        bool writable;             /* Whether this page is
writable. */
};

        Also, each page has four types, as it is enumerate.
enum splmt_page_type{
        FILE, SWAP, ZERO, FRAME
};
        FILE: This page contains a file.
        SWAP: This page is in swap slot.
        ZERO: This page is a zero page.
        FRAME: Normally, loaded in frame table.


---- ALGORITHMS ----

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

        Given the page, we could certainly have its user virtual
address,
and with user virtual address, we would be able to search in the
hash
table to find the page table entry, with a constant time complexity.
In the structure of page table entry, there is a member variable
that
point to the frame address that we can use to locate the frame.

>> 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?

        There are two member variables in the structure of
supplemental
page table entry. So the structure of page table entry work as a
connecter of two sides to update, thus avoid the issues.
```

```
---- SYNCHRONIZATION ----

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

        Declear a frame table lock as global variable. Every time
allocate
new frame table entry, try to acquire the lock first. In this we, we
could guarantee that there is only one process allocating new frame,
and thus avoid the data races.

---- RATIONALE ----

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

        The hash table has many advantages, such as the quick
search
time complexity, which is almost a constant. It is suitable to
implement the page table, as the hash table need a distinct key
to work, and virtual memory indeed has the user virtual address
which is distinct, thus make it proper to set to be the key value.
Also, there are many cases that we want to learn the frame entry
or physical address given the user virtual address, so the hash
table is the best choice, in terms of performence.

                        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.

struct list frame_table;  /* The frame table. */

struct frame_table_entry{
        void* frame_addr;              /* The frame addr of entry.
*/
        void* user_addr;               /* User addr of the entry.
*/
        struct thread* owner;          /* The thread which own the
page.*/
        struct lock frame_entry_lock;  /* Lock to synchronize. */

        struct list_elem elem;        /* Elem to put it into the
list. */
        struct splmt_page_entry* spte; /* Corosponding page table
entry. */
};
```

---- 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 used a clock algorithm to choose the frame to evict.
When it
failed to allocate new page, it iterate through all entries in the
frame table. If a page is accessed, give it a second chance, set the
access boolean value to false, and if there is a page with accessed
boolean value false, then this is the page to evict.

>> 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?

        After the eviction process, the corrosponding page in the
page table
is adjusted(i.e. set the bool in_swap to true, type SWAP, and
is_loaded
false). Then the process Q will be able to learn from page table
that it
no longer has the page and it is in swap slot now.

>> 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.

We need to check two parts before growing stack. The first is check
whether
the page address is within the stack space, which is between the
physical
base - max stack size and the physcal base. And the second is to
check whether
the fault page is below the stack pointer, or the page fault 4 and
32 bytes
below the stack pointer, if any is satisfaid, then we can grow 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.)

        The most trouble of synchronization come from the process
of
frame declearation and eviction, so we use a frame table lock to
guarantee that only one process is possibly add new entry to frame
table. So it would gracefully handle most of the trouble, like it

is mentioned in the following quistions. Also, we used another
lock for swap process, mainly in order to deal with the block
device synchronization.

>> 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?

        Declear a lock as global varibale for the swap process, every time
when some pages need to swap out to the block device, or page need to
be swapped in from the swap slot, the process need to acquire the swap
lock firstly. With the help of this lock, we could be able to make sure
that only one process is possibly doing the eviction as well as
reclaimation process.

>> 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?

        The lock for the frame table would solve this problem, as the
process of reading from file system or swap slot as well as eviction
are all encapsulated in the process of allocating new frame table
entry.
        If a process need to read from file system or swap, it will need
to allocate a new frame table entry. Before the process of allocation,
the process P need to acquire the frame table lock. Only if process P
acquired the frame table lock, would it be able to read from file
system or swap. So it is for eviction of process Q. So with the help
of frame table lock, we could ensure that they will not interfere.

>> 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?

Since I use lazy load, if the page faults happens, I need to bring
in
pages pages. And the original fault page may be in the file system,
or in a swap slot, or it might simply be an all-zero page. And I
first
check its type, and then install and load page from file, or swap in
a
page from swap, or set a zero page.
And for invalid virtual addresses, I first check whether it is
present
or not, if it is not present, I will terminate immediately or exit
the
process, determinating on it is user to not.

---- 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.

As it is illustrated in the question B5, the majority of
trouble
of synchronization come from the frame table entry. So we used a
frame table lock to deal with most of issues. Also, the read and
write
of the block device as well as the bitmap also need to synchronize,
so a lock is also needed. But this part of synchronization seems to
be
irrelevant to the part we mentioned above, so we used two different
locks.

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.

```
/* Structure of the memory map owned by a process. */
struct process_mmap{
  struct list_elem elem;  /* Elem to put into the list. */
  struct file* file;      /* A pointer to the mmap file. */
  void *addr;             /* the virtual address the file map to. */
  int id;                 /* the size of the mapped file. */
};
```

---- 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.

Each process own a list of mapped files, each map contains its
address in memory,
the mapped file and the map id. The system call map the file open as
fd into the
virtual address space of the process, and then the entire file is
mapped into
consecutive virtual pages starting at address, which can record the
files directly
in memory of the process.
Page fault: the swap page is swapped in the frame from swap slot,
and the others
are brought in filesystem.
Eviction: we need to write the file back if it is dirty for other
pages, but swap
page can just be swapped out into the swap slot
and does not need to
write back.

Non-file related pages are moved to a swap partition upon eviction,
regardless of whether or not the page is dirty.

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

Before start mapping, first find whether the range of pages mapped
overlaps any existing set of mapped pages, if true, it will return
-1 and stop the mapping procedure. Only after checking that no
overlap happen, will it start mapping.

---- 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.

Although some of their implementation can be shared, I did not share
the code to avoid potential problem that may occur. And I also build
a structure of the memory map owned by a process, to store some
information of the mmap.

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?