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

---- GROUP ----

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

Yining She sheyn@shanghaitech.edu.cn

Shaoting Peng pengsht@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.

Online sources:

1. https://wenku.baidu.com/view/24c27f23dd36a32d73758185.html
2. https://www.doc88.com/p-3837935351748.html

Offline sources: 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 `page.c`:

```
struct sup_page_table_entry
{
    // user virtual address
    void * upage;
    // physical virtual address
    void * kpage;
    // if the page entry is swaped into disk, we need to store the index
    size_t swap_index;
    // which thread keeps this supplementary page table
    struct thread * owner;
    // where the page is currently
    enum page_status status;
    struct hash_elem hash_elem;

    // for filesys
    struct file *file;
    // file page offset
    off_t file_offset;
    // bytes within one page
    uint32_t read_bytes, zero_bytes;
    bool writable;
};
```

In `thread.c`:

```
struct thread
{
  ...
  // use a hash table to store sup_page_table_entry
  struct hash sup_pt;
  ...
}
```

---- ALGORITHMS ----

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

We write this code in `load_segment()` :

```
struct thread *curr = thread_current ();
ASSERT (pagedir_get_page(curr->pagedir, upage) == NULL);
if (! supt_set_page_from_filesys(&curr->sup_pt, upage, file, ofs,
      page_read_bytes, page_zero_bytes, writable))
{
  return false;
}
ofs += PGSIZE;
```

This is the 'lazy_load' implementation. First we use `pagedir_get_page()` to check if upage is already mapped. If not, we use `supt_set_page_from_filesys()` to set upage into SPT.

To access the data stored in the SPT, we use the following function:

```
struct sup_page_table_entry *
sup_pte_lookup (struct hash *sup_table, const void *address)
{
  struct sup_page_table_entry p;
  struct hash_elem *e;

  p.upage = address;
  e = hash_find (sup_table, &p.hash_elem);
  return e != NULL ?
    hash_entry (e, struct sup_page_table_entry, hash_elem) : NULL;
}
```

Given a SPT and an address, we can simply use `hash_find()` to get a hash entry, which is a SPTE.

> 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 stored accessed and dirty bits in our frame table and use `pagedir_is_accessed()` and `pagedir_is_dirty()` to get those bits.

---- SYNCHRONIZATION ----

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

To get a new frame, our implementation use `palloc_get_page()` . And in this function, there's a code segment as following:

```
...
lock_acquire (&pool->lock);
page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);  lock_release
(&pool->lock);
...
```

The `bitmap_scan_and_flip()` function is used to record a frame's status, and there's a lock to ensure that this operation is atomic, so the races can be avoided.

---- RATIONALE ----

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

We use **hash table** to store **sup_page_table_entry**.

1. If this question aims to ask why do we use hash table, it's because this hash table could be very big, and hash operations can support quick finds.

2. If this question aims to ask why do we use sup_page_table_entry, it's because we have to store the data of a page when it's lazy loading, or when a frame is currently in swap, `pagedir_get_page()` would return NULL, but we can still find it using our implementation.

```
            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 `frame.c`:

```
// a frame table to store frame_table_entries
static struct list frame_table;
// lock on frame operations
static struct lock frame_lock;
```

In `frame.h`:

```
struct frame_table_entry{
    // physical user address
    void * kpage;
    // virtual user address
    void * upage;
    // which thread this frame belongs to
    struct thread * owner;
    struct list_elem elem;
    // accessed and dirty bits
    bool accessed;
    bool dirty;
};
```

---- 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 write a function using clock policy called `frame_to_evict()`:

```
frame_to_evict (void *upage)
{
    struct list_elem *e;
```

```
        struct frame_table_entry *frame = NULL;
        for (int i=0; i<2; i++){
            for (e = list_begin (&frame_table); e != list_end (&frame_table);
                 e = list_next (e))
            {
                frame = list_entry (e, struct frame_table_entry, elem);
                if (upage == frame->upage)
                {
                    continue;
                }
                if (pagedir_is_accessed (frame->owner->pagedir, frame->upage))
                {
                    pagedir_set_accessed
                      (frame->owner->pagedir, frame->upage, false);
                }
                else
                {
                    return frame;
                }
            }
        }
        return frame;
    }
```

It's obvious to understand: traverse the `frame_table` list, find a frame with accessed bit unset, then evict this frame. Also, as it advance, it clears the accessed bit if it's not 0.

> 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?

This is done in our function `get_free_frame()` :

```
    /*  Get a new user frame from user pool.
        If there is no free frame, evict an old frame. */
    void *
    get_free_frame (void *upage)
    {
        // lock_acquire (&frame_lock);
        bool success = true;
        void *kpage = (void*)palloc_get_page (PAL_ZERO | PAL_USER);
        struct frame_table_entry *frame = NULL;

        // if current frame is full, choose one to evict
        if (kpage == NULL)
        {
            struct thread *curr = thread_current ();
            // palloc() returns NULL, means we need to evict some frame
            frame = frame_to_evict ();
            kpage = frame->kpage;
            ASSERT (frame != NULL);
            // printf(">>> In get_free_frame, frame to evict is %x\n", frame);
            size_t swap_index = swap_out (frame->kpage);

            struct sup_page_table_entry *spte;
            spte = sup_pte_lookup(&frame->owner->sup_pt, frame->upage);
            // store the info about the frame into sup_pt
            set_sup_swap (spte, swap_index);
            // clear the kpage
            pagedir_clear_page (frame->owner->pagedir, frame->upage);
            palloc_free_page(kpage);

            kpage = palloc_get_page (PAL_ZERO | PAL_USER);
            ASSERT (kpage != NULL);
            // memset (frame, 0, PGSIZE);
            // frame->kpage = NULL;
            // frame->owner = curr->tid;
            // printf(">>> swap success, index:%d\n", swap_index);
        }
        frame = frame_table_set_page (kpage, upage, frame);
        // lock_release (&frame_lock);

        return kpage;
    }
```

If `kpage == NULL`, we find a frame to evict. Once found, we look up the corresponding spte in sup_pt by `sup_pte_lookup()`, and store the info about the frame into sup_pt by `set_sup_swap()`.

Then, we use `pagedir_clear_page()` and `palloc_free_page()` to delete every piece of information about that frame to make it fresh as original.

Fianlly, we are able to set the newly get frame into `frame_table` by `frame_table_set_page()`.

> 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 check this in `exception.c`:

```
bool on_stack = (fault_addr > esp || fault_addr == esp-4 || fault_addr == esp-32);
```

In `page_fault()`, if the fault_address is 32 bytes lower than esp, it will be identified as an invalid virtual address because of the PUSHA and PUSH instruction, otherwise the stack will be extended by one page.

---- 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.)

In our implementation, we must consider that there are activities between processes and between frames and pages. We use the pin bit to help our synchronization with general usage of locks or semaphores.

The pin bit helps avoid eviction since our eviction algorithm also ignores entries with pin bit set to true. However, when the kernel is done accessing that frame, the pin bit is set back to false. Consequently, that entry is able to be evicted.

> 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?

In our implementation, the pin bit will show that whether a frame is being modified.

In the sence that Q is working on its own frame, and P needs to do work with the frame table. The pin bit is set to true since Q is modifying that frame; so our eviction algorithm would ignore frames with true pin bit. P can't evict Q's frames. When P is trying to evict a frame, the load bit is set to false, so Q must reload that page so the page can be faulted back in. In this way the race condition is avoided.

> 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?

It's also done by pin bit. Q should ignore any pin bit frame that is true in the eviction algorithm (i.e. Q are not allowed to evict those frames) because the pin bit is set during P's operations with the frame. Our design is to avoid page faulting as much as possible, but when a fault occurs we have to load a page in to avoid the next access to fault.

> 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?

Before any syscall, we use `is_user_vaddr_valid()` to check whether the adddress is valid:

```
bool is_user_vaddr_valid (void* addr, struct intr_frame *f){
  if (addr == NULL || !is_user_vaddr(addr) || addr < 0x8048000)
    return false;
  struct thread * cur = thread_current ();
  bool success = pagedir_get_page (cur->pagedir, addr);
  if (!success){
    if (addr > f->esp || addr == f->esp-4 || addr == f->esp-32){
      // grow stack!
      void *upage = pg_round_down (addr);
      if (sup_pte_lookup (&cur->sup_pt, upage) != NULL)
        return true;
      void *kpage = get_free_frame (upage);
      success = pagedir_set_page (cur->pagedir, upage, kpage, true);
    }
    else{
      struct sup_page_table_entry *spte;
      void *fault_page = pg_round_down (addr);
      spte = sup_pte_lookup (&cur->sup_pt, fault_page);
      if (spte != NULL)
        return sup_load_page (&cur->sup_pt, cur->pagedir, fault_page);
      else
        return false;
    }
  }
  return success;
}
```

Then we try `pagedir_get_page()` to check if actually in physical frames. If it's not, we check for virtual address to determine whether to do stack growth or simply to load it back if it's already in `sup_pt` using `sup_pte_lookup()` to get the corresponding spte and use `sup_load_page()` to load it back.

If attempt to access to invalid virtual addresses, this function returns `false` and cause the thread to exit directly.

---- 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 choose 2 locks: `file_lock` as it does in project2 and `frame_lock`. We use lock every time we read or write to the file or frame_table, parallelism is limited but can avoiding the mistake by synchronization.

```
                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 `thread.h`:

```
  // used for memory mapped file's id
  typedef int mmapid_t;
```

```
// similar to file_node in proj2
struct mmf_node
{
   // mapping id for mmf
   mmapid_t mid;
   // number of pages in an mmf
   int page_num;
   // file pointer
   struct file* file_ptr;
   struct list_elem elem;
   // user virtual address for mmf
   void *addr;
};

struct thread
{
    ...
    // a list of memory mapped files
    struct list mmf_list;
    ...
}
```

---- 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.

Every thread has a list called `mmm_list` . In `mmap` syscall, we load a file into memory, call `allocate_mid()` to get a mid for this file, store the basic information in `mmf_node` and push it into current thread's `mmf_list` .

In `munmap` syscall, we use a `for` loop to find the `mmf_node` according to the given mid:

```
for (e = list_begin (mmfiles); e != list_end (mmfiles); e = list_next (e))
{
  struct mmf_node *mmfile = list_entry (e, struct mmf_node, elem);
  if(mmfile->mid == mid)
  {
    ...
  }
}
```

Then, we delete each page entry in SPT, write the content at this address back into the file (if `pagedir_is_dirty()` return true) using `file_write()`, free the corresponding spte and close the file.

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

In `mmap()`, we first calculate the number of pages in the file, and traverse each user page while checking for existent mapping in `spt` and `pagedir` :

```
if (sup_pte_lookup
      (&curr->sup_pt, upage) || pagedir_get_page(curr->pagedir, upage))
{
  // printf(">>> sup_pt already has this entry!\n");
  return -1;
}
```

If there's an overlap, we simply return -1 and terminate this syscall.

---- 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.

It's true that much of their implementation can be shared because they have a lot similarities. But our implementation still use two functions independently: `supt_set_page_from_MMF()` and `supt_set_page_from_filesys()` because they are used in different scenarios: one in `mmap()` syscall while the other one in `load_segment()` to achieve lazy load. Though they look similar, we think it would be less confusing and easy to debug.

```
               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?