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

---- GROUP ----

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

Group 19
韩嘉恒 hanjh1@shanghaitech.edu.cn
李鹤洋 lihy3@shanghaitech.edu.cn

Contributions:
Discuss together, and coding part:
Managing the frame table, managing the swap table and implementation of putting all together in ‘process.c’ by Han Jiaheng.
Managing the page table, memory mapped files by Li Heyang.

---- 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.

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.h:

```
enum page_status
{
  MEMORY, /* In memory. */
  SWAP,   /* At swap. */
  FILE,   /* Map to file. */
  LAZY    /* Lazy loading. */
};
struct sup_pt_elem
{
  uint32_t *vaddr;              /* Virtual address of the SPTE. */
  struct thread *owner;         /* The thread that owns the SPTE. */
  bool dirty;                   /* Dirty bit. */
  bool accessed;                /* Access bit. */
  struct hash_elem hash_elem;   /* Hash element defined in hash.h */
  enum page_status status;      /* Current status of the SPTE. */
  uint32_t swap_id;             /* Swap id of the SPTE. */

  struct file *file;            /* Owning file of the SPTE.  */
  off_t ofs;                    /* Offset of the file. */
  uint32_t read_bytes;          /* Read bytes of the file. */
  uint32_t zero_bytes;          /* Zero bytes of the file. */
  bool writable;                /*  Writable bit. */
  struct list_elem elem;        /* List element defined in list.h. */
};
```

In thread.h:

```
/* Project 3: Virtual Memory*/
   struct hash sup_pages;                /* A hash table records all SPTEs that this
thread allocated. */
```

---- ALGORITHMS ----

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

If we want to find a SPTE in the SPT, the function in page.h :
`struct sup_pt_elem *find_pt_elem(struct thread *t, const uint32_t *vaddr);`
is used to achieve this utility. You just need to pass in the virtual address as the vaddr, and it will return the found SPTE to you if exist, or NULL if doesn’t. We use a hash table to represent a page table, so each SPTE is a hash element in the hash table. Hence you can do operations and accessing the data to this SPTE stored in the SPT by finding it using vaddr. Besides its all information, there is a “status” in it to keep track of where it comes from, such as MEMORY, FILE, SWAP or for lazy loading.

>> 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?

Accessed and dirty bits are used when eviction and write back. In our implementation, all kernel code uses the kernel virtual address to access the page, also the kernel will never change the content in the page. Only user will do that. So, the dirty bits of the user virtual address can be used when determine whether a page need to be actually written back to swap or file and the dirty bit of the kernel address and kernel' s access to that address will not affect this procedure.

---- SYNCHRONIZATION ----

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

In function void *palloc_get_frame(enum palloc_flags flags) of frame.c,
We have sema_down(&frame_sema); in the beginning and sema_up(&frame_sema); in the end, hence we can prevent from racing using semaphore and avoid conflicts when two user processes both need a new frame at the same time.

---- RATIONALE ----

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

The system has provided a 2-level linked list for keeping pages, level-1 is page directory and level-2 is page table. And our frame table is a linked list, because we seldom search for a frame and linked list is just right for our implementation for it' s simple and easy, and more complicate data structure is not necessary. And our supplemental page table use a hash table in each process to keep all the supplemental pages in that process, and the virtual address is the key to be hashed to find corresponding supplemental page in the hash table.

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

```
/* The frame table, stored as a list. */
static struct list frame_table;
/* Semaphore to avoid racing in frame operation. */
static struct semaphore frame_sema;
```

In frame.h

```
struct frame_elem
{
  /* Used in list frame_table. */
  struct list_elem elem;
  /* The physical page(in kernal address space). */
  uint32_t *page;
  /* Allocation time of the frame. */
  int64_t time;
  /* The thread owns the frame. */
  struct thread *owner;
  /* The virtual page(in user address space). */
  uint32_t *vaddr;
};
```

In swap.c

```
struct swap
{
  /* Hardware interface to access SWAP. */
  struct block *swap_block;
  /* Semaphore to avoid racing in swap operation. */
  struct semaphore swap_sema;
  /* Bitmap to record whether a block is used or free. */
  struct bitmap *swap_pool;
  /* Extra infomation to avoid rewrite clean page back. */
  struct info *info;
};
struct info
{
  /* The owner thread of the block. */
  struct thread *owner;
  /* The coorisonding user address of the block. */
  void *vaddr;
};
static struct swap *swap;
```

---- 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.

The frame table is a list and each time a new frame is required, it will be pushed at the end of the list. We choose the first allocated existing frame as the evict frame, and in our implementation, it is the first element of the frame table. It is a efficient O(1) algorithm to choose a evict and this could also enforce fairness.

>> 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?

There are two things to do. One is to install the frame in process P’ s page table and supplemental page table, calling pagedir_set_page () and update the information in the supplemental page table element of that virtual address.
The second thing to do is to notify thread Q that frame Q is no longer in memory. This is done by calling pagedir_clear_page() and set status in the supplemental page table element to FILE or SWAP for stack and mmap respectively. The next time process Q access frame Q will result in a page fault instead of reading process P’ s data.

>> 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.

According to the document, 80*x*86 PUSH operation may cause a page fault 4 bytes below the stack pointer, so we accept it as a valid stack growth if the fault address is above 4 bytes below the stack pointer and it is below PHYS_BASE, also we give each process at most 2048 pages for stack, otherwise the fault address will be invalid and the process will be determined.

---- 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.)

Holding while waiting is one requirement for deadlock to happen. So, we avoid holding while waiting. In our implementation, if a process acquires a lock successfully, it should always successfully continue running until it finishes the critical section and release the lock.

>> 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?

When an evict is chosen, the first thing to do is to call pagedir_clear_page() and if process Q access the evict frame, a page fault will happen. The page fault handler will try to allocate a new frame and at that time, process Q will be block by the frame_sema until process P successfully write the frame to swap, then process Q can load the page back to memory and the SWAP operation can avoid the race.

>> 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?

In the page fault handler, when a page need to be read from file system or swap, the handler will first allocate a frame for the page, but don't install it to the frame table, which means when choosing evicts, the frame is not in the candidate. So it couldn't be evict from memory. When the page successfully read from file or swap to memory, the frame will be installed in the frame table and everything go back to normal.

>> 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?

In our implementation, for everything except the system call, we use the page fault handler to bring paged-out pages back to memory from swap or files, this includes both user programs and kernel code except system calls. In system calls, because the issue of the interrupt frame, we did not directly access those paged-out memory and invoke a page fault handler. Instead, we will first check whether pointers pointing to the arguments are valid, including pointer to pointer, and if we find a paged-out page, we will bring it back to memory, which is similar to what happens in page fault handler.

---- 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.

Take frame operations and swap operations as an example. To ensure synchronization, whenever a swap operation allocate or free some block, there will be a lock to ensure only one process is changing the bitmap used in the swap. Also, there is a lock in the frame operation to do the same. However, if frame operation and swap operation call each other' s function, deadlock may happen. Fortunately, only frame operation will call swap operation' s function and thus no holding while waiting will happen. This gives the idea that if we can divide the VM system into some layers and higher layer only call functions of lower layer in its critical section, no deadlock will happen and it would be safe. This also allow some parallelism, e.g. page fault handler can also call swap directly with out calling frame function.

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 syscall.c:

```
struct mmap_struct
{
  int mmapid;                 /* Mmap id of the mapping. */
  struct list_elem elem;    /* List element for constructing list. */
  struct list maped_pages; /* List that records all the pages used for this map. */
  uint32_t *addr;           /* Start address of the file. */
};
```

In thread.h:

```
struct list maped_files; /* A list records all the mapped files by this thread. */
int mmapid_flag;         /* A counter for generating mmap_id and unmap when exit
the process. */
```

---- 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 keeps a list of mapped files, and when the system do the system call ‘mmap’, the file will be mapped into the memory and return a map_id of it. And unmap will free the corresponding memory and write back to the file if relative pages are dirty, then set page invalid. And also, when the process exits, it unmap all the files it had mapped and free the memory. And for the difference between swap pages and other pages, as we have defined a `page_status` in ‘page.h’, we can determine whether it comes from FILE or SWAP. So if it comes from FILE, page fault handler would load from file and eviction would write back to the file. And if it comes from SWAP, page fault handler would load from swap and eviction would write back to swap.

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

Before mapping the file into our virtual memory system, we check whether it would overlap any existing segment. Firstly, we get the length of the file and calculate how many pages are needed for our mapping. If there are *num_pages* of pages needed for our map, we check from *addr* to *addr* + *PGSIZE* * *num_pages,* step is *PGSIZE,* to see whether these pages are already in our page table or not. If already exist, exit right away with return value -1.

---- 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.

In our implementation, the code for these two situations shares a lot. Based on no conflicts and confusion happen, some structure of our SPTE are used in both “mmap” and those of data demand-paged from executables:

```
  struct file *file;            /* Owning file of the SPTE.  */
  off_t ofs;                    /* Offset of the file. */
  uint32_t read_bytes;          /* Read bytes of the file. */
  uint32_t zero_bytes;          /* Zero bytes of the file. */
```

This part are shared in our implementation in two situations mentioned above, because they all need to keep the data of file and relative information such as offset, read_bytes and zero_bytes. They have the similar use and hence we share them in our implementation for simpler code structure and higher performance.

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?