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

---- GROUP ----

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

Zijun Xu <xuzj@shanghaitech.edu.cn>
Runze Yuan <yuanrz@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.
https://github.com/codyjack/OS-pintos

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 `vm/page.h'

```
/* Supplementary page table entry type determined by the file type in it. */
enum spte_type
{
  _FILE,    /* Normal file type, no mmap or swapping */
  _SWAP,    /* The file is already stored on swap area because of eviction */
  _MMAP     /* If the file is memory mapped, set spte to this type */
};

/* Supplementary page table entry for hash table storing it */
struct page_suppl_entry
  {
    struct file* file;       /* Store file pointer for loading */
    enum spte_type type;     /* Type of the file */

    uint8_t *upage;          /* User virtual address, unique identifier for
                                a page, key for hash table */
    off_t ofs;
    uint32_t read_bytes;     /* Bytes at UPAGE must be read from file
                                starting at OFS */

    uint32_t zero_bytes;     /* Bytes at UPAGE + READ_BYTES must be zeroed. */
    bool writable;           /* False if the page is read-only */
    bool loaded;             /* True if the page is loaded to physical addr */

    size_t swap_idx;         /* Index on swap bitmap returned by swap_out() */
```

```
    struct lock pte_lock;    /* For pte synchronization */
    struct hash_elem elem;   /* Hash table element */
  };
```

In `userprog/thread.h`

```
struct thread {

    struct hash suppl_page_table;       /* Supplementary page table */
}
```

---- ALGORITHMS ----

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

When we are ready to execute a exe file, we need to first load the content of the file into memory. This is done by `load_segment()` function in `process.c`. In project2, we load all the content into memory in the beginning. But now, we divide this file into several pages, and malloc several pte to store the corresponding information such as file pointer, the data bytes in this page, the offset to the beginning of the file, etc. Then insert these ptes into the supplementary page table without loading these. This is called load lazily.

When a page fault occurs, we can get fault address which is the upage, then we can use this upage to find the corresponding pte in supplementary page table. If it is found, then we need to load this page from the exe file. The information we stored in the pte can lead us to load right data in the file.

1. Allocate a new frame, if memory s full evict one.
2. We start to load at the position `ofs` and load `read_bytes` bytes, and set the rest bytes which is `zero_bytes` to zero. Data of this page goes into the allocated frame.
3. Then map the frame to the pte's upage. This process is to map physical memory to virtual memory, using `install_page()` function.

Note these pte's type is `FILE`

Another place to create pte is `stack_grow()` function. When a thread touch the place past the stack pointer, the stack needs to grow. Then we need to malloc a new pte and a new frame, then map and install them. Note that, this pte does not have a file pointer, because it is just for stack, and no data in any file will be loaded, so the type of this pte is `SWAP`.

>> 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 avoid the issue by avoid the problem by only accessing user data through the user virtual address.
In our design, we never use kernel virtual address to touch user data.

---- SYNCHRONIZATION ----

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

Using frame_lock to ensure mutual exclusion when adding new frames.

---- RATIONALE ----

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

We need a frame table to record all pages mapped on physical memory since we need eviction on this project, so we would like to decide which frame to evict. We also need a supplementary page table, since we apply lazy loading policy on this project, and we let the page fault handler to allocate frames, so we shall record all the related information for a page when loading to physical memory.

We use a linked list for frame table, since when dealing with physical address, we often need to insert (allocate frame) or delete (eviction) a frame, these operations take O(1) time on a linked list. However, finding a frame needs O(n) time for a linked list, but we only need to iterate over the whole list when performing eviction and free operations, these won't happen a lot.

We use a hash table to store all supplementary table entries with the user virtual address as the key. Since we need to find a supplementary table entry in the table when page fault happens, and page fault happens all the time in this project since we apply the lazy load policy, so we need a fast way to find an entry from the table. Therefore, we use hash table which need only O(1) time on average to extract a specific element.

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 list frame_table;
struct lock frame_lock;

struct ft_entry
  {
    void *frame;
    struct page_suppl_entry *pte;
    struct thread *owner;
    struct list_elem elem;
  };
```

In `vm/swap.h`

```
struct block *swap_block;  /* The block disk for swapping space */

struct bitmap *swap_map;   /* Bitmap to indicate a swap slot is used or not */

struct lock swap_lock;     /* Lock to protect the operations on swap_map */
```

---- 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 a linked list to record all the frames we allocated.

What we used is clock algorithm. The whole process is
  1.Interate on this list to get every ft_entry.
  2.Since we record pte and owner process in this ft_entry struct, we can use
    pagedir_is_accessed() function to determine if this frame has been
    accessed.
    a.If this frame has been accessed, set the accessed bit to false and
      continue iterating.
    b.If this frame has not been accessed, we get a frame which can be evicted.
      If this frame is for mmap file, check the dirty. If it is true, write
      back to the corresponding file.
      If it is not, we need to use swap function to swap the data in this frame
      to swap slot for temporary. Then change the type of this page to `SWAP`.
      Now we get a evicted frame, memset it to all zero and update the other
      information about this frame. For example, remove the old ft_entry from
      the frame table, create a new pte and a new ft_entry, remap this pte to
      evicted frame, insert the new ft_entry into the frame table, etc.
      After maintaining all the information, we can return this evicted frame.
  3.If every frame has been accessed, since we set its accessed bit to false
    in procedure 2. Iterate this table again, then we can evict successfully.

>> 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?

As stated above, when evcited, we delete the ft_entry corresponding to process
Q, and insert a ft_entry corresponding to process P. Then in frame table, it
indicates that this frame belongs to process P rather than process Q.

Also, in evicting process, we use function `pagedir_clear_page()` to mark the
upage corresponding to this frame to "not present" which will be checked in
page_fault_handler.

The two method guarantees that this frame belongs to process P rather than
process Q.

>> 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.

In page fault handler, we use fault_addr to find pte firstly, if it is not
found, then it probably needs stack growth.

After that we need to check the invalidation of the fault address.
This address should be within the stack size, so it needs to lager than
PHYS_BASE - STACK_LIMIT. The STACK_LIMIT we set defaultly is 8MB.
Then because of the `PUSH` and `PUSHA` instructions, we check if this
fault_addr is larger than esp-32. This also checks if the fault address is
within stack.

If the conditions above are not satisfied, we probably meet the conditions
such that the stack is full or it touched the kernel memory, etc. At this point
the process needs to terminate.

---- 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 every file operations, such as open, write, etc, we need to add lock to guarantee that this operation is atomic. Note that in project2 and project3, we will both use file operations, so it should use the same lock.

In swap_in() and swap_out(), every time we use bitmap_scan_and_flip() function we need to acquire a swap lock to ensure that no other process can modify the bitmap at the same time.

When we do operations of the frame table, such as palloc_get_frame(), evict_frame(), etc. We need to acquire the frame lock to ensure that only one process is modifying the frame table.

>> 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 we evict a frame, we can get the corresponding pte through ft_entry, so we can acquire the pte_lock in pte. Every time we modify or read the frame, we need to acquire the corresponding pte_lock to make sure that the process can not be interrupted by other processes's operation on this frame. In other word, a frame can only be touched by one process at one time.

When P is evicting one frame, the lock is acquired, if Q is trying to modify or read the frame, it must acquire the lock first, but Q find that P has acquired the lock, so it will get to sleep until Q finishes evicting the frame.

We will acquire the pte_lock when invoking the load_file or load_swap functions So, if P is evicting and Q tries to page back in, Q will go to sleep until P finishes and then Q can get the page back.

We notice that the above two conditions will greatly lower the performance. Since a frame may swap in immediately after evicting.

>> 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?

When P is causing a page to be read from the file system or swap, as stated above, it will acquire the page's lock. If Q is trying to evict this page, Q will go to sleep until the reading process is done. After that, Q can evict this page.

This method will definitely lower the performance but is working, since the eviction can only be done after reading. However, the probability of the condition is actually small.

A better solution would be using pinning, but we didn't implement it because of the complexity.

>> 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?

During syscalls, we will first check the validity of function arguments of esp before calling the syscalls functions. This check only check if the pointer is user address, if not we will exit(-1). While accessing user memory, we shall prevent them from occurring. Take syscall_read() for example, we do not let page fault happen during reading to buffer, we handle it manually before the page fault handler deals with it. We also ensure that all the file operations are atomic, since we add a fs_lock to all file operations.

---- 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 frame table lock to ensure the frame table can not be modifed by multiple threads, this is essential especially in evict_frame() function since we must guarantee that during eviction the frame table can not modified.

The operation in bitmap must also be atomic, since a process using swap_in can interrupt a process using swap_out. Then a swap lock is needed.

If we use a global lock, then swap operation and evict operation will not appear at the same time, which will cause low parallelism.

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.

```
/* Under 'thread.h' */
/* These two new data structures added to thread are similar to the
   fd_list and file_num added to thread for project 2 */
struct thread
  {
    //....
    struct list mmap_list; /* List for all memory mapped files in a thread */
    mapid_t mmf_num;       /* Unique number to identify each mapped file */
  }

struct mmap_entry
  {
    mapid_t mmap_id;       /* Unique number to identify a mapped file */
    void *uvaddr;          /* Unique user virtual address for first page */
    struct file *file;     /* File pointer */
    unsigned int page_num; /* Number of pages the mmap file will have */
    struct list_elem elem; /* List element for mmap_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.

When a user call the syscall mmap(), we will first validate that mapping, such as the validation of mapping addr. If the mapping passed the check, we will reopen a new file pointer and malloc() an mmap_entry to store the file pointer and the required information. The mmf_num of the current thread will increase, and the new mmap_entry will have a unique mmap_id under the current thread. Then we will call page_lazy_load() to create new SPT entries for this mmap file and insert them to the SPT. Finally we insert the mmap_entry to mmap_list and return the corresponding mmap_id if the mapping is successfully created.

When a user call syscall munmap(), we will iterate the whole mmap_list and find the corresponding file by mmap_id and free all the related resources, like the created SPT entries and frames if allocated, then delete it from the mmap_list. Note that if the mmap file write to some pages, we will write back the content back to the file.

The page fault handler treats swap pages and other pages with a slight difference. For both situations, the page fault handler will allocate new frames and install the page. However, for swap pages, the data is get from swap slots but for other pages(data or mmap files), data is get from files.

There is no difference between swap pages and other pages when deciding which frame to evict. All the frames can be evicted, and the data will be swap out the corresponding swap slots. The spte_type of any evicted page will be changed to _SWAP after eviction. For mmap files, if its pages are dirty, we will write its data back to file. For all kinds of pages, page is uninstalled and an empty frame will replace the original.

>> C3: Explain how you determine whether a new file mapping overlaps
>> another segment, either at the time the mapping is created or later.

We implement this before the mapping is created, since under this situation a mmap() syscall should immediately fail. Since we can get the file length of the file we are going to map, then we can use a loop to check if the user addr starting from ADDR to (ADDR + file length) is already mapped by other pages or there is already a page table entry occupying that address space. We use pagedir_get_page() and page_hash_find() to do this check. If one of the two situations happens, we immediately return -1 to fail this 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.

In our implementation, they do share much of the code. In SPT entry, they share all the member values except their spte_type is different, one is _FILE and the other is _MMAP. This slight difference is demanded during eviction, we need to write back _MMAP file content to their original files from a dirty page. For data demand-paged from executables, they are written to swap area and their spte_type will changed to _SWAP after that. Other values they share are all needed when we apply our lazy load policy by calling page_lazy_load() and page_load_file(). Therefore, we share much of the code for the two situations

except the spte_type for the demand of frame eviction.

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?

Some questions under "PAGING TO AND FROM DISK -> synchronization" is repeated and some questions are hard to answer if students do not implement such design. The tests of Pintos do not gives such check on synchronization. Therefore, we cannot know if our meets the requirement but the questions are still there. So I wonder if some of the questions will be optional depends on students' actual design.

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

>> Any other comments?