Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Yixin Yang (杨艺欣) yangyixin@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

All tests specified in the Lab 3a documentation have passed (tested for multiple times).

> 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.

ChatGPT was used for debugging suggestions and design ideas.

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.

```c
/* vm/frame.h */
typedef void * kpage_t; /* Kernel page (physical address) */
typedef void * upage_t; /* User page (virtual address) */

/* Frame table entry */
struct frame_entry
  {
    struct hash_elem h_elem;   /* Hash element (keyed by kpage) */
    struct list_elem l_elem;   /* List element for global frame list */
    kpage_t kpage;               /* Kernel (physical) address of frame */
    struct thread *owner;      /* Owning process/thread */
    upage_t upage;               /* User virtual address mapped here */
    bool pinned;               /* If true, do not evict */
  };

struct lock frame_lock;        /* Lock for frame table */

/* Initializes the global frame table (called in thread_init). */
void frame_init (void);

/* Allocates a user frame for user page 'upage', evicting if needed.
   Called in load_page_by_spt () */
kpage_t frame_alloc (upage_t upage);

/* Frees a frame when a process exits (unmaps and releases).
   Called in destroy_spe () */
void frame_free (kpage_t kpage);

/* Pins/unpins a frame to prevent/allow eviction. */
void frame_set_pinned (kpage_t kpage, bool pinned);

/* vm/page.h */
/* PAGE_BIN: Backed by a file (e.g., executable)
   PAGE_ZERO: Zeroed page (e.g., stack, heap)
   PAGE_SWAP: Swapped out page (e.g., evicted from memory) */
enum page_type { PAGE_BIN, PAGE_ZERO, PAGE_SWAP };

/* Supplemental page table entry */
struct spt_entry
  {
    struct hash_elem h_elem;
    upage_t upage;               /* User virtual page (key) */
    enum page_type type;
    struct file *file;         /* Backing file (for PAGE_BIN) */
    off_t ofs;                 /* Offset in file */
    size_t read_bytes;         /* Bytes to read */
    size_t zero_bytes;         /* Bytes to zero */
    block_sector_t swap_slot; /* Swap slot index if PAGE_SWAP, else -1 */
    bool writable;
    struct lock spte_lock;     /* Lock for this entry */
  };

/* Initialize supplemental page table (called in process creation). */
void spt_init (struct hash *spt);

/* Create and insert a new supplemental page table entry for file-backed
   page upage. */
bool spt_install_bin_page (struct hash *spt, upage_t upage,
                           struct file *file, off_t ofs,
                           size_t read_bytes, size_t zero_bytes,
                           bool writable);

/* Create and insert a new supplemental page table entry for zeroed page
   upage. */
bool spt_install_zero_page (struct hash *spt, upage_t upage,
                            bool writable);
```

```c
/* Destroy the supplemental page table, freeing supplemental page
   table entries, swap slots, frame table entries and kernel pages. */
void spt_destroy (struct hash *spt, struct lock *spt_lock);

/* Set swap slot in the supplemental page table entry. */
void spt_set_page_swapped (struct hash *spt, struct lock *spt_lock,
                           upage_t upage, block_sector_t swap_slot);

/* Called on page fault. Load page according to supplemental page table
   entry. */
bool load_page_by_spt (void *fault_addr);

/* vm/frame.c */
static struct hash frame_map;    /* Hash table: key = kpage */
static struct list frame_list;   /* List of all frames for eviction */
static struct list_elem *clock_hand;  /* Clock pointer */

/* Hash functions for frame table. */
static unsigned frame_hash (const struct hash_elem *e, void *aux UNUSED);
static bool frame_less (const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED);

/* Choose a victim frame using clock algorithm (skip pinned).
   The victim frame will be pinned. */
static struct frame_entry *pick_victim_frame (void);

/* vm/page.c */
/* Hash functions for supplemental page table. */
static unsigned page_hash (const struct hash_elem *e, void *aux UNUSED);
static bool page_less (const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED);

/* Lookup SPT entry by user page (NULL if not found). */
static struct spt_entry *suppagedir_find (struct hash *spt, upage_t upage);

/* Destroy a supplemental page table entry.
   Helper function for spt_destroy (). */
static void destroy_spe (struct hash_elem *e, void *aux UNUSED);

/* userprog/syscall.c */
/* Called in syscall_read () and syscall_write () to pin the frames in user buffer. */
static void page_set_pinned (const void *buffer, unsigned size, bool pinned);

/* userprog/process.c */
/* Only create and insert entries in the supplemental page table,
   does not actually read file content into pages. */
static bool lazy_load_page (size_t read_bytes, size_t zero_bytes, struct file *file, off_t ofs, uint8_t *upage, bool writable);
```

```c
/* userprog/process.c */
struct proc_info
  {
    /* ... */

    /* VM */
    struct hash sup_page_table;    /**< Supplemental page table. */
    struct lock spt_lock;          /**< Lock for the sup_page_table. */
  };
```

ALGORITHMS

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

There are two cases for accessing the SPT data:

The first case is when a not-present page fault occurs. The faulting address is passed to `load_page_by_spt()`, which does the following:

1. Checks if the user virtual page corresponding to the faulting address is present in the SPT. If not, it returns false (which will cause the page fault handler to print an error message and kill the process).
2. If the page is in the SPT, locks the corresponding SPT entry, makes a copy of the entry (to avoid race conditions for reads), and unlocks the entry. Subsequent reads to the entry will be done using the copy. Only writes will require acquiring and releasing the SPT entry lock.
3. Allocates a kernel page using `frame_alloc()`, which uses `palloc()` to allocate a frame from the user pool, or evicts one if `palloc()` fails. The allocated frame is pinned to prevent eviction.
4. Loads the page:
    - If the page is zeroed, `memset()` it with zeroes.
    - If the page is file-backed, reads the file content into the page.
    - If the page was swapped out, reads the content from the swap slot.
5. Installs the page into the page table using `pagedir_set_page()`. Sets the dirty bit if the page is zeroed or was swapped out.
6. Unpins the frame.

The second case is when `frame_alloc` evicts a dirty victim frame. After the frame is written back to a swap slot, `frame_alloc` will call `spt_set_page_swapped()` to update the SPT entry of the evicted page, indicating that it is now swapped out to a certain swap slot.

> 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?

I avoided the issue by only accessing the page table by user virtual addresses.

The only exception is in `load_page_by_spt()`, where I need to access the page table by kernel addresses. It does not count as accessing or dirtying the page, since the page is not yet mapped to the user process.

For swapped-out pages and zeroed pages, the dirty bit of the user virtual page is manually set after the page is installed into the page table.

SYNCHRONIZATION

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

There is a `frame_lock` acquired in the beginning and released at the end of `frame_alloc()`. So only one process at a time can allocate a frame.

The allocated frame will be pinned until it is installed into the page table, this prevents it from being evicted while loading still in progress.

RATIONALE

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

SPT:

- It is a hash table, which allows for fast (near $O(1)$) lookup of page entries. There are often many pages in the SPT, so a hash table is more efficient than a list or array.
- The SPT is stored in the process's `proc_info` struct, protected by a table lock `spt_lock`. Each entry in the SPT has an entry lock `spte_lock` to protect the entry itself. This allows more fine-grained locking, enabling higher parallelism.

Frame table:

- We use a hash table and a list to store frame table entries. The hash table is used to look up the corresponding frame table entry by its kernel page. The list is used by the eviction algorithm to pick a victim frame.
    - The reason to use hash table is the same as in SPT: fast lookup.
    - The clock algorithm needs to iterate over all frame entries. Lists provide a much more straightforward way to do this than hash tables. Plus, it is easier to make a circular list than a circular hash table.
- The frame table is protected by a single lock `frame_lock`.

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.

Data structures already included in the previous section are skipped.

```
/* vm/swap.h */
/* Initialize swap subsystem (bitmap) */
void swap_init (void);

/* Write a frame’s data to swap, returning its slot index. */
block_sector_t swap_write (void *frame);

/* Read a page from swap slot into frame. */
void swap_read (block_sector_t slot, void *frame);

/* Free a swap slot. */
void swap_free (block_sector_t slot);

/* vm/swap.c */
static struct bitmap *swap_table;  /* Bitmap of swap slots, false = free */
static struct lock swap_lock;      /* Lock for swap table access */
static struct block *swap_block;   /* Swap block device */

/* Number of sectors in a page. */
static const size_t SECTORS_PER_PAGE = PGSIZE / BLOCK_SECTOR_SIZE;
```

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.

I implemented the clock algorithm in `pick_victim_frame()`:

1. `frame_list` is a circular list of all frame entries. The `clock_hand` pointer points to the current frame to be checked.
2. Once `pick_victim_frame()` is called, it will check the frame entry pointed to by `clock_hand`:
    i. If the frame entry is pinned, skip it
    ii. Otherwise, check the accessed bit of its corresponding user page entry:
        - If the accessed bit is set, clear it.
        - If the accessed bit is not set, this frame is a picked for eviction. Pin and return it.
3. Move `clock_hand` to the next frame entry in the list and repeat step 2.

> 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?

1. Clear the page table entry of the obtained frame. Now any access by Q to it will cause a page fault.
2. If the corresponding page is dirty, write it back to a swap slot by `swap_write()`, and update the corresponding SPT entry to indicate that the page is now swapped out to a certain swap slot, by calling `spt_set_page_swapped()`.
3. Update the frame entry to indicate that it is now owned by P, mapped to P's user page. It will be pinned until it is installed into P's page table.

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.)

Locks corresponding to Lab 3a:

- SPT table lock `spt_lock` (per process): protects the SPT of a process.
- SPT entry lock `spte_lock` (per entry): protects the SPT entry of a process.
- Frame table lock `frame_lock` (global): protects the frame table.
- Swap lock `swap_lock` (global): protects the swap system.
- Filesystem lock `filesys_lock` (global): protects the filesystem.

Deadlock happens when **nested** locks are acquired in **inconsistent orders**. Based on this, I carefully designed the synchronization logic:

- Nested locks are avoided at best:

    - In function `load`, `filesys_lock` is released before lazy segment loading (which does not touch the file system), and re-acquired after. This prevents the nesting of `filesys_lock` and `spt_lock` (which is acquired in `lazy_load_page()`).
    - In `load_page_by_spt()`, although all five locks are used, none of them are explicitly nested. This is achieved mainly by carefully lock releasing and **making a local copy of the shared SPT entry**:
        - After a frame is allocated by `frame_alloc()`, the SPT table lock `spt_lock` is used to lookup the SPT entry. Once the entry is found, `spt_lock` is released immediately and the entry lock `spte_lock` is acquired. Then, a copy of the entry is made and `spte_lock` is released. Any subsequent reads to the entry will be done using the copy, and only writes will require acquiring and releasing the SPT entry lock.
- There are three instances of lock nesting in my whole Lab 3a code:
    - `spt_lock` and `spte_lock` acquired independently inside critical section of `frame_lock`, in `frame_alloc()`.
    - `swap_lock` inside that of `frame_lock`, in `frame_alloc()`.
    - `spte_lock` inside that of `spt_lock`, which in turn inside that of `frame_lock`, in `spt_destroy()`.
    - By ensuring consistent lock acquisition orders, my code is deadlock-free.
        - `frame_lock -> spt_lock -> spte_lock`
        - `frame_lock -> swap_lock`

I ran `page-parallel` test for more than 35 times, and my submission code passed all of them.

> 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?

After P picks Q's frame, **before the evicting process starts**, P will clear Q's page table entry, so that any access by Q to it will cause a page fault.

In `load_page_by_spt()`, a new frame is allocated **before** looking up and copying the SPT entry. This ensures that when Q runs into a page fault for this frame while the eviction has not be done, it will immediately be blocked by the `frame_lock` in `frame_alloc()`, so there will be no race condition for Q's SPT entry.

> 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?

Frames allocated by `frame_alloc()` are pinned until they are installed into the page table (which happens after the read).

> 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?

Page faults are used to bring in pages.

For `read()` and `write()` system calls, the frames in the user buffer are pinned to prevent eviction during the write/read process (and unpinned after).

In the page fault handler, if the page is not present, it will first try to load the page by looking it up in the SPT. If that fails, it will check if it should extend the stack (not implemented yet in my Lab 3a submission). Then, if the processor is in user mode, it will print an error message and kill the process. Otherwise, it must be an invalid access in system calls, so it will set eax to 0xffffffff (as the return value -1) and copies its former value into eip (as specified in Lab 2 doc).

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.

In my first design, there was no SPT entry locks. But during the development process, I encountered countless synchronization issues, which forced me to start over multiple times, adding and then removing entry locks. Eventually, I achieved a correct implementation without entry locks, but to

improve parallelism, I added entry locks again.

Apart from improving parallelism, I chose to use multiple locks to wrap synchronization logic inside functions, making the code more modular. For example, the swap system is internally synchronized by `swap_lock`, so code using it does not need to worry about acquiring and releasing `swap_lock`, which makes implementation cleaner and less bug-prone.