Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Jin Yang 2000012935@stu.pku.edu.cn

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

> Linux source code about momery management.
> x86 page management.

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.

```
enum page_status
{
    PAGE_IN_FRAME, /**< Already mapped.*/
    PAGE_ALLZERO,  /**< All zero.*/
    PAGE_IN_SWAP,  /**< In swap space.*/
    PAGE_IN_FILE,  /**< In file. */
};
```

> The status of a page.

```
struct sup_page_table
{
    struct hash page_map;
};
```

> The supplyment page table for each thread.

```
struct sup_page_table_entry
{
    void *upage; /**< Virtual address of the page. */
    void *kpage; /**< Kernel frame of the page, only accessable when status is
IN_FRAME.*/
```

```
    enum page_status status; /**< Status of the page. */

    struct file *file;   /**< File to store the page. */
    off_t offset;        /**< Offset of the file. */
    uint32_t read_bytes; /**< Bytes already read. */
    uint32_t zero_bytes; /**< Bytes need to be set 0. */

    swap_index_t swap_index; /**< Swap index of the page, only accessable when
status is IN_SWAP.*/

    bool writable; /**< Is the page writable. */
    bool dirty;    /**< Dirty bit. */

    struct hash_elem helem; /**< Hash element. */
};
```

> A supplyment page table entry, records neccessary infomation for the unloaded page.

```
struct thread{
    ...
    #ifdef USERPROG
    /* Owned by userprog/process.c. */
    void *current_esp;          /**< Stack pointer.*/
    struct sup_page_table *spt; /**< Supplyment page table. */
    ...
};
```

> **current_esp** is used in handling page fault. **spt** is the supplyment page table for the thread.

ALGORITHMS

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

> The function **spt_find()** does the job. Since I use a hash table to store the data, function **spt_find()** calls **hash_find()** to find the SPTE about a given page. If can't find, it returns NULL.

> 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?

> For the accessed bit, I don't store it in the SPTE and only handle it with the function in pagedir.h, so there is no problem. As for the dirty bit, it is stored in the SPTE and only accessed in function **spt_set_dirty()**, which is only called in function **frame_alloc()**. In **frame_alloc()**, the following code is to ensure coordinate access:

```
bool dirty = pagedir_is_dirty(pd, upage) || pagedir_is_dirty(pd, kpage);
spt_set_dirty(spt, upage, dirty);
```

SYNCHRONIZATION

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

> I use a frame_lock in frame allocation.Each time, only one process can do frame allocation.

RATIONALE

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

> After we have lazy loading and swap pages, we need more infomation to handle page table. So it's reasonable to create a supplyment page table for each thread.

> For the loaded pages, the original page table is enough. But for the unloaded ones, including pages in files and swapped pages, we need supplyment page table to manage them.The original page table uses user virtual address to locate the physical frame, while the supplemental page table use user virtual address to locate things on the disk.

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 frame_table_entry
{
    void *kpage;          /**< Physical address of the page. */
    void *upage;          /**< User virtual address of the page. */
    struct thread *owner; /**< The owner thread of the frame. */
    bool pinned;          /**< Whether the frame will be evicted.*/

    struct list_elem lelem; /**< List element. */
    struct hash_elem helem; /**< Hash element. */
};
```

> The frame table entry, handles the map between virtual address and physical address.

```
static struct hash frame_map;
```

> The frame table for whole system.

```
static struct list frame_list;
```

> The frame list for eviction algorithm.

```
static struct lock frame_lock;
```

> The frame lock to ensure synchronization.

```
static struct list_elem *clock_ptr;
```

> The global clock algorithm pointer.

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 use the adapted clock algorithm called second chance algorithm, which is introduced in the class.

> Traverse the whole **frame_list**, a pointer indicates which frame to be replaced next. When a frame is needed, the pointer advances until it finds a frame with a 0 accessed bit. As it advances, it clears the accessed bits. If a frame is chosen, it will be removed from the **frame_map** and **frame_list**.

> For some vital pages and newly-loaded pages, they are pinned so that they will never be evicted.

> 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?

> In the frame table entry, the owner thread is stored. So when evicting a page, function **pagedir_clear_page()** is called to adjust the 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.)

> The frame table and swap space is global, while the supplyment page table is owned by each thread.

> For the reason above, I use a gloabal frame lock and swap lock. When handling file operations, a global file lock is used.

> I provide APIs for those locks, ensuring acquiring one lock at one time and they don't interact with ohter locks, which means no deadlock for the locks above.

> One more thing about the file lock is that I didn't find **process_execute()** also use file systems and didn't use the file lock in syscall **exec()** before. It is fixed now.

> 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?

> The frame lock is used when allocating a frame. If P is evicting Q's frame, the frame lock will be acquired first, So when Q access the page, a page fault occurs and Q tries to bring the page back using **frame_alloc()**, and Q will be held at acquiring the lock until P's allocation finishes.

> 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?

> A newly-allocated page is pinned until **load** successfully in **start_process()**, so it won't be evicted until then.

> Also, since I use second chance algorithm, a newly allocated page will be pushed back into the list. It's unlikely to choose and evict a newly allocated page

> 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 page fault handler, if a page is swapped out, it will be swapped in. And the pinned page will not be swapped out.

> In syscalls, the function **check_valid()** will be called to test whether the address is valid. An invalid address exits the current thread. And also a mechanism to determine an invalid access is use in page fault, which kills the process if page fault is triggered by user (In project2, a page fault triggered by syscall(kernel) will exit with -1).

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.

> To use only one lock for the whole vm system limits the parallelism, so I introduce a separate lock for each global resource. And those locks are internal so they don't influence each other, which means no deadlock.