Lab 3a Design Doc

Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Name: Zhang Lingxin

Email: 2100013018@stu.pku.edu.cn

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

Before writing this document after submitting project 3a, I have finished the whole project 3 and modified some relevant implements when writing it. As a result, some details in the submitted version of project 3a may be different from those in the document. I have submitted my project 3b, so the modified source code can be viewed there.

> 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://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/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.

```
/* frame.h */
#define FRAME_MAGIC 0xcd7bae4c

struct hash frame_hash;
struct lock frame_lock;

struct frame_table_entry{
    struct thread *owner;       /**< The owner thread of the frame. */
    void *kpage, *upage;        /**< The kernel/user page. */
    struct hash_elem elem;    /**< Hash element. */
    bool pinned;                /**< Is the frame pinned? */
    unsigned magic;             /**< Detect stack overflow */
};

/* page.h */
#define PAGE_MAGIC 0xc0610feb

enum page_status{ALL_ZERO, IN_FRAME, IN_FILESYS, IN_SWAP};

struct supp_page_table_entry{
    struct hash_elem h_elem;

    void *upage;                /**< User page. */
    void *kpage;                /**< kernel page. */

    /* For pages from files */
    struct file* src_file;      /**< The source file(if any). */
    off_t offset;               /**< File offset. */
    uint32_t read_byte;         /**< The number of bytes read from file. */
    uint32_t zero_byte;         /**< The remaining bytes filled with zero. */
    size_t swap_idx;            /**< Swap index for swap slots tracking. */
```

```
    enum page_status page_stat; /**< The status of current page.*/

    bool writable;              /**< Is the page writable? */
    bool dirty;                 /**< Is the page dirty? */

    unsigned magic;             /**< Detects stack overflow. */
};

/* thread.h */
struct thread{
  ...
    struct hash supp_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.

A function, `supp_page_to_spte(struct thread *t, void *upage)`, can get the corresponding `supp_page_table_entry` of a given page `upage` in the supplementary page table of thread `t`. The supplementary page table is implemented as a hash map, so the function `hash_find()` is used.

```
/* Search the supplementary table of thread T for
   the SPTE of UPAGE */
struct supp_page_table_entry *
supp_page_to_spte(struct thread *t, void *upage){
    struct supp_page_table_entry tmp;
    tmp.upage = upage;
    struct hash_elem *e = hash_find(&t->supp_page_table, &tmp.h_elem);
    return e == NULL ?
           NULL : hash_entry(e, struct supp_page_table_entry, h_elem);
}
```

> 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?

Take the dirty bits as an example. As we know, we can access the dirty bits of user/kernel pages in `pagedir` with the function `pagedir_is_dirty()`. Besides, dirty bit is kept in `struct supp_page_table_entry`.

If a function wants to know the actual dirty status of a frame, it accesses `struct supp_page_table_entry` to get the corresponding `upage`, `kpage` and the dirty bit stored in the structure. Then it gets the actual dirty bit with `OR` operations :

```
dirty = pagedir_is_dirty(t->pagedir, spte->upage) ||
        pagedir_is_dirty(t->pagedir, spte->kpage) ||
        spte->dirty;
```

SYNCHRONIZATION

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

There is a global lock for frames, `frame_lock`, to make sure that only one process can access any code in `frame.c` in a time. So when two user processes both need a new frame at the same time, a thread will have to wait.

```
/* a typical function in frame.c */
struct RET_TYPE
frame_function_demo(struct PARA_TYPE){
  lock_acquire(&frame_lock);
  /* Do something */
  lock_release(&frame_lock);
  return ANS;
}
```

RATIONALE

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

I choose to create a hash map, `frame_hash`, to store the frame table.

```
struct frame_table_entry{
    struct thread *owner;       /**< The owner thread of the frame. */
    void *kpage, *upage;        /**< The kernel/user page. */
    struct hash_elem elem;    /**< Hash element. */
    bool pinned;                /**< Is the frame pinned? */
    unsigned magic;             /**< Detect stack overflow */
};
```

This is because a hash map supports fast insertion, lookup and deletion, which is necessary in frame management. And a hash map simplifies the design by reducing unnecessary traversals of frame table.

Besides, using a hash map is not more difficult than a list because most of necessary functions are provided in `hash.h`. In fact, I don't see any reason to use a list at all.

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.

```
#define SECTORS_PER_PAGE 8 // = PGSIZE / BLOCK_SECTOR_SIZE = 4096/512 = 8

struct block *swap_block;
struct bitmap *swap_map;
/* swap_map: 0 for available, 1 for unavailable */

size_t swap_size;
```

Others are listed in Part A1.

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 ***clock algorithm*** in frame eviction.

The function traverses the frame table to check and update the access status of a frame.

- If a frame is pinned, ignore it.
- if the access bit is 1, reset it.
- If the access bit is 0, choose the frame.

Every frame is checked no more than twice in clock algorithm, so the iteration time is set to `2*size`.

If there is not an available frame (e.g. all the frames are pinned), return `NULL`, which will panic the kernel later.

```
/* Choose a frame to evict. Clock algorithm. */
struct frame_table_entry* frame_pick_to_evict(struct thread *t){
    size_t size = hash_size(&frame_hash);
    uint32_t* pagedir = t->pagedir;

    struct hash_iterator i;

    /* Clock algorithm */

    /* Go through the frame table list. We will always find a
       evictable frame in no more than 2*size iterations (some frames may
       be pinned).*/

    bool first_loop = 1;
LOOP:

    hash_first (&i, &frame_hash);
    while (hash_next (&i))
    {
        struct frame_table_entry *fte =
            hash_entry (hash_cur (&i), struct frame_table_entry, elem);
        ASSERT(is_frame_table_entry(fte));

        /* If a frame is pinned, ignore it.
           if the access bit is 1, reset it;
           If the access bit is 0, choose the frame.*/
        if(!fte->pinned){
            if(!pagedir_is_accessed(pagedir, fte->upage))
                return fte;
            pagedir_set_accessed(pagedir, fte->upage, 0);
        }
    }

    if(first_loop){
        first_loop = 0;
        goto LOOP;
    }

    return NULL;
}
```

> 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?

- Remove the supplementary page table entry from the supplementary page table of Q.
- Call `pagedir_clear_page()` to remove the corresponding page (virtual address) from `pagedir` of process Q.

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 VM synchronization design has only one new lock: `frame_lock`, a global lock for frame synchronization. Whenever a thread tries to access frame table, it has to acquire the lock first.

There is no other resource shared between processes in my implement of project 3. Supplementary page table is implemented as per-thread, which is safe as long as the synchronization between thread is implemented correctly. Swap slots are only used in frame eviction, which is safe as long as `frame_lock` is implemented correctly. (File system synchronization are implemented in project 2 with `filesys_lock`)

So the only chances of deadlock is when `frame_lock` and `filesys_lock` are both acquired by the same thread, which can happen in frame from files and some system calls. I prevent deadlock by make sure that always acquire the `filesys_lock` first and release `filesys_lock` later.

> 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 global lock for frame synchronization, `frame_lock`, makes sure that a thread must acquire the lock before accessing any information in the frame table.

P will acquire the lock before eviction. So Q cannot access or modify the page or bring a page back by page fault because it has to acquire the lock first, which is held by P.

> 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?

The global lock for frame synchronization, `frame_lock`, makes sure that will not happen.

As mentioned above in B6, a thread must acquire the lock before accessing any information in the frame table. As reading a page from file system or swap will modify the information stored in the frame table, P will acquire the lock first. So Q can never interfere by frame eviction or any similar operations.

> 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?

I use page faults to bring in pages as in project 2.

The page address is checked before get into the real functionality of the system call. If it’s in paged out, it will be brought back. Then the corresponding frame to the page is set as unpinned. After that, the page will be safe from eviction.

Every address will also be checked for validity when checking paged-out pages. If we find an invalid address, the process will be terminated as in project 2.

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.

The VM synchronization design has only one new lock: `frame_lock`, **a global lock for frame synchronization**.

This is because most resources are per-thread in my design. Supplementary page table is implemented as per-thread, which is safe as long as the synchronization between thread is implemented correctly. Swap slots are only used in frame eviction, which is safe as long as `frame_lock` is implemented correctly.

Although `frame_lock` is global and may limit parallelism, I try to make the critical sector as small as possible to allow for better parallelism.

I choose the design because making most of the resources as per-thread will prevent a lot of synchronization problems and reduce debugging workload.