Lab3a Design Doc

Project 3a: Virtual Memory

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/frame.c

```
 /*A list of frame table entry used for eviction*/
static struct list frame_list;

/*A mapping from user virtual address to kernel virtual address.*/
static struct hash frame_hashmap;

/*A global lock, to ensure critical sections on frame operations.*/
static struct lock frame_lock;

/*A pointer pointing to the next frame to be checked for swapping.*/
static struct list_elem* clock_pointer;

struct frame_table_entry
{
   void *upage;      /*User virtual page UPAGE.*/
   void *kpage;      /*Kernel virtual address KPAGE*/
   struct list_elem l_elem;        /*Defined for frame_list.*/
   struct hash_elem h_elem;     /*Defined for frame_hashmap.*/
   struct thread *t;  /*The corresponding thread*/
   bool pinned;      /*Used to prevent a frame from being evicted, while it is acquiring
   some resources.If it is true, it is never evicted.*/
};
```

In vm/page.h

```
 /** States of pages owned by the process. */
enum page_status
{
   PAGE_FRAME,      /*In frame.*/
```

```
    PAGE_SWAP,       /*In swap slot.*/
    PAGE_FILESYS,    /*Stored in filesystem.*/
    PAGE_ALLZERO,    /*All-zero page.*/
};

/** Supplemental page table entry.*/
struct supple_table_entry
{
    enum page_status status;        /*Page's state. */
    void *upage;                    /*User virtual page UPAGE.*/
    void *kpage;                    /*Kernel virtual address KPAGE/

    /*Information of file to be loaded.*/
    struct file*file;
    off_t offset;
    uint32_t read_bytes, zero_bytes;
    bool writable;

    bool dirty;    /*Dirty bit*/
    int swap_index;    /*Swap bitmap index*/
    struct hash_elem elem;    /*Defined for thread's supplemental page table.*/
};


/**  Supplementable page table.Each process owns a supplemental page table.*/
struct supplemental_page_table
{
    struct hash page_hashmap;
};
```

In thread.h

```
 struct thread{
#ifdef VM
    struct supplemental_page_table* supple_table;  /**< Supplemental page table. */
#endif
}
```

ALGORITHMS

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

I store every page's entry in its process's `supple_table->page_hashmap` ,whose hash function is defined as follows:

```
static unsigned supple_hash_func(const struct hash_elem *e, void *aux UNUSED)
{
    struct supple_table_entry*supple_entry = hash_entry(e,struct supple_table_entry,elem);
    return hash_bytes(&supple_entry->upage,sizeof(supple_entry->upage));
}
```

So we can find each page's supplementable page entry through page's starting user virtual address.
An example:find the SPT entry of the page whose user virtual address is `upage` in supplemental page table `stable`.

```
struct supple_table_entry supple_tmp_entry;
supple_tmp_entry.upage = upage;
struct hash_elem* e = hash_find(&stable->page_hashmap,&supple_tmp_entry.elem);
struct supple_table_entry*supple_entry = hash_entry(e,struct supple_table_entry,elem);
```

`supple_entry` is a pointer pointing to the page's `supple_table_entry`.So we can use this pointer to access the data stored in the SPT about the given page.

> 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?

(1)Accessed bits
In my implementation,I only use user virtual page's accessed bit to support my implementation of frame eviction policy and don't use or modify kernel virtual page's accessed bit.
(2)Dirty bits
For dirty bit, I add a member `dirty` to `struct supple_table_entry` to manage the modification of dirty bit.
Pintosbook claims the alises between kernel and virtual address:

> that is, two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated.

When we load page in memory,we've built an alise between kernel and user virtual address.There are the steps I've taken to handle alises.
(a)Set the `dirty` to false when the SPT entry is created.
(b)When evicting a page from frame to swap slot,obtain user and kernel virtual addresses's dirty bits and stored it in SPT entry's `dirty` member as shown below.

```
struct thread* prev_t = evicted_entry->t;
bool is_dirty = false;
is_dirty = is_dirty || pagedir_is_dirty(evicted_entry->t->pagedir, evicted_entry->upage);
is_dirty = is_dirty || pagedir_is_dirty(evicted_entry->t->pagedir, evicted_entry->kpage);

/*Modify the evicted frame's supplemental page table entry.*/
vm_supple_frame_to_swap(prev_t->supple_table,evicted_entry->upage,swap_index,is_dirty);
```

```
bool vm_supple_frame_to_swap(struct supplemental_page_table *stable, void *upage,
int swap_index,bool dirty)
{
    struct supple_table_entry *supple_entry = vm_supple_lookup(stable,upage);
    ASSERT(supple_entry != NULL);

    supple_entry->status = PAGE_SWAP;
    supple_entry->swap_index = swap_index;
    supple_entry->kpage = NULL;
    supple_entry->dirty = dirty;
}
```

(c)When loading page into memory(frame),we will built a new alise between user virtual page and a new kernel virtual page.It should be pointed out that the new kernel virtual page still keeps the message corresponding with the previous user virtual page.So we need to **set the kernel virtual page's dirty bit to false**.

SYNCHRONIZATION

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

Whether a user process gets a new frame through `palloc_get_page` or through evcition policy,once the frame is obtained by a user process,the frame is pinned so that this frame is unavailable to any onther processes asking for a new frame.

RATIONALE

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

In struct `supple_table_entry` and struct `frame_table` ,there are two members `upage` and `vpage` to represent virtual-to-physical mappings.I think it is the most intuitive and understandable way to think and implement.

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 swap.c

```
/** The block where the swap slot is located. */
static struct block*swap_slots;

/** A swap bitmap records which slots are used and which are available.1 represents that the swap slot is in use,0
represents empty swap slots. */
static struct bitmap*swap_bitmap;
```

```
/** The number of sectors a page will occupy. */
static const int SECTOR_NUM = 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 use clock algorithm to choose a frame to evict.
I maintains a list of page frames(`frame_list`) in memory, and keeping track of each page's "second chance" status,which is represented by user virtual page's accessed bit.

When a new frame is needed, the clock algorithm scans through `frame_list` , looking for a page that has not been referenced recently. The `clock_pointer` points to the frame we are going to check.
The steps I take to find the proper frame is as follows:
(1)If the frame is pinned,it can't be chosen for eviction, so we just skip it and check the next frame.
(2)If the frame's `upage` 's accessed bit is 1,it is referenced recently, so we set `upage` 's accessed bit to 0 and go to check the next frame on `frame_list` .
(3)If the frame's `upage` 's accessed bit is 0,we choose this frame to evict.
(4)If until the end of `frame_list` we don't find a proper frame to evict,jump to the start of `frame_list` and scan it again.

> 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?

For Q:
Remove references to the frame from any page table that refers to it.
(1)This frame's entry's member `void*upage` will be set to P's page's user virtual address and `struct thread* t` will point to P's `struct thread` .
(2)Call `pagedir_clear_page` on Q's `pagedir` and the user virtual address of the evicted page to remove reference.
(2)The SPT entry of Q's evicted page should be modified.

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.)

(1)Lock
`frame_hashmap` and `frame_list` can be accessed and modified by all user processes.
So for synchronization, a user process should acquire `frame_lock` before it modifies these two structures or frames' entries and finally releases `frame_lock` when the access ends.
(2)Pinned

When a frame is to be evicted to swap slot or the page on swap slot or file system is to be read into frame,the frame should be pinned to protect itself from being acquired by any other user process .

How to prevent deadlock:
In my VM synchronization design,I carefully deal with the operations that need synchronization and only use lock in frame.c.
(1)Each user process has its own SPT,so for SPT we don't worry about synchronization.
(2)When modifying frames,use `frame_lock` to achieve mutual exclusion.
(3)When modifying `swap_bitmap` and `swap_slots` ,although they can be accessed by many user processes simultaneously,they don't need additional synchronization operations.Because when evicting a frame to swap slot in `vm_frame_allocate` (before we call `vm_swap_out_slot`),the user process has already got `frame_lock` .For other modifications on `swap_bitmap` and `swap_slots` ,there is no need for synchronization.

> 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?

(1)How do you ensure that Q cannot access or modify the page during the eviction process?
Before we call `vm_swap_out_slot` to evict the the frame to swap slot,we call `pagedir_clear_page` on Q's `pagedir` and the page.So any access to Q's page during eviction process will cause page_fault.

(2)How do you avoid a race between P evicting Q's frame and Q faulting the page back in?
We won't change page's SPT entry's `status` from `PAGE_FRAME` to `PAGE_SWAP` until the eviction process ends.So although the access to Q's page during eviction process will cause page_fault, `page_fault` will do nothing,because `vm_load_page` only load a page to frame when page's SPT entry's `status` is not `PAGE_FRAME` .

> 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 `vm_frame_allocate` ,when we allocate a frame for a user page,we will pin it.So when the user process gets a frame through `vm_frame_allocate` ,it is already pinned.In `vm_load_page` ,only when the work of reading from the file system or swap is done will we unpin the frame.At the same time, as we mentioned in B2,we avoid choosing a pinned frame to evict.

> 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?

(1)In all system calls before Lab3b, I just use page faults to bring in pages and don't do anything special.

(2)How to handle attempted accesses to invalid virtual addresses
I handle attempted accesses to invalid virtual addresses in system call handler and page fault handler.
If the virtual address is not user virtual address,just kill this process.Then if the virtual page is not in

frames,page fault handler will take control and either load the page into frame if the address is valid or kill the process if the virtual address is invalid.

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.

My VM synchronization design is to use a single lock.It is easy but limit parallelism.
If I want to improve parallelism,I need to break `frame_list` and `frame_hashmap` into small pieces and coordinate multiple locks,also I need take the synchronization of swap table into careful consideration.The design will be much more complicated.I don't have enough time and ability to achieve this.