Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Xinle Cheng adacheng@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.

https://github.com/JJHWAN/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 `frame.h`

```
struct list lru_list;

struct list_elem* lru_clock;

struct frame{
    void *kaddr;
    struct spt_entry *spe;
    struct thread *t;
    struct list_elem lru;
};
```

1. **struct list lru_list;**
    - List of frames to implement the Least Recently Used (LRU) eviction policy.
2. **struct list_elem* lru_clock;**
    - Pointer to the current position in the LRU clock algorithm.
3. **struct frame { void *kaddr; struct spt_entry *spe; struct thread *t; struct list_elem lru; };**
    - Represents a frame with physical address, supplemental page table entry, owning thread, and LRU list element.

in `page.c`

```
#define BIN 0
#define FILE 1
#define ANON 2

struct spt_entry{

    uint8_t type;
    void *vaddr;

    bool writable;
    bool is_loaded;

    struct file* file;

    size_t offset;
    size_t read_bytes;
    size_t zero_bytes;
    struct hash_elem elem;
    struct list_elem mmap_elem;

    size_t swap_slot;
};
```

1. **#define BIN 0**
    - Identifier for binary page type.
2. **#define FILE 1**
    - Identifier for file-backed page type.
3. **#define ANON 2**
    - Identifier for anonymous (non-file-backed) page type.
4. **struct spt_entry { uint8_t type; void *vaddr; bool writable; bool is_loaded; struct file* file; size_t offset; size_t read_bytes; size_t zero_bytes; struct hash_elem elem; struct list_elem mmap_elem; size_t swap_slot; };**
    - Represents an entry in the supplemental page table with type, virtual address, and related metadata.

ALGORITHMS

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

Given a page, I call `insert_spe` in `load_segment` and `setup_stack`. I store the type of the page, (one in `BIN` , `SWAP` , `FILE`) and many other meta data in the `SPT`.

I call `find_spe` in `page_fault`, finding the SPT corrresponding to a certain fault address. I use `hash_find` to search for the SPT given a thread and a fault address.

> 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?

In function `evict_pages`, I check the access bit and the dirty bit, if `pagedir_is_accessed(page->t->pagedir, page->spe->vaddr)` returns true, I set the access bit to false and let this page skip its turn during the **clock algorithm**. If `pagedir_is_dirty(page->t->pagedir, page->spe->vaddr)`, I write the changed data to the file the page belongs to.

For the issue above, I just let the user address to access the dirty bit and the access bit.

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

A global lock `vm_lock` is maintained to avoid this race. Each time I operate the `lru_list` , the lock is needed.

RATIONALE

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

The SPT is easier to implement other than storing extra meta data on PTE itself. Moreover, the SPT is clean and easy to read.

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.h`

```
static struct block *swap_block;

static struct bitmap *swap_bitmap;

struct lock swap_lock;
```

1. **static struct block *swap_block;**
    - Pointer to the block device used for swap space.
2. **static struct bitmap *swap_bitmap;**
    - Bitmap to track used and free swap slots.
3. **struct lock swap_lock;**
    - Lock to synchronize access to the swap space.

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.

The `evict_pages()` function attempts to free pages by iterating through the LRU (Least Recently Used) list and checking the accessed bit of each page. If a page is not accessed, it either writes back to the file or swaps out to the swap space, clearing the page and marking it as evicted.

To be more specific:

As for the clock algorithm, I iterate through the LRU list, if the page has been accessed recently, I clear the accessed bit and continue to the next page.

```
if (pagedir_is_accessed(page->t->pagedir, page->spe->vaddr)) {
    pagedir_set_accessed(page->t->pagedir, page->spe->vaddr, false);
    continue;
}
```

After that, if the page is dirty or an anonymous page, decide between writing it back to the file or swapping it out.

```
if (pagedir_is_dirty(page->t->pagedir, page->spe->vaddr) || page->spe->type ==
ANON) {
    if (page->spe->type == FILE) {
        lock_acquire(&file_lock);
        file_write_at(page->spe->file, page->kaddr, page->spe->read_bytes, page-
>spe->offset);
        lock_release(&file_lock);
    } else {
        page->spe->type = ANON;
        page->spe->swap_slot = swap_out(page->kaddr);
    }
    page->spe->is_loaded = false;
    pagedir_clear_page(page->t->pagedir, page->spe->vaddr);
    lock_release(&vm_lock);
    break;
}
```

After finding the page to evict, I remove the page from the LRU list and update the LRU clock if necessary.

```
lock_acquire(&vm_lock);
if (lru_clock == &page->lru) {
    lru_clock = list_remove(lru_clock);
} else {
    list_remove(&page->lru);
}
lock_release(&vm_lock);
```

> 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 for the Q frame, I use `*pagedir_clear_page(uint32_t* *pd, void* upage)` to mark user virtual page UPAGE "not present" in page directory PD. Later accesses to the page will fault. I also remove the frame from the LRU list and free the corresponding physical address of the frame.

I change its type to *ANON* if it has been swapped out to the swap slot.

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.)

I use a `swap_lock` and a `file_lock` to enable synchronization. The `swap_lock` enforces synchronization on the swapping process, and `file_lock` enables synchronization when writing and reading files.

> 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?

If the Q's frame is evicted and `pagedir_clear_page` has been called, then later we access Q, it will cause a `page_fault`.

Meanwhile, I first get the `vm_lock` during the eviction, making sure that Q cannot access or modify the `lru_list`.

Also I'll get the `file_lock` during the eviction if `page->spe->type == FILE` and the page is accessed, so Q's operation to the file will be blocked.

> 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?

While P is reading, it holds the `file_lock`, if Q attempts to evict P, it has to get the `file_lock` too, (since P is marked as FILE), and thus the race is avoided. P will first read all the data it needed.

> 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?

When doing system calls and the corresponding page is not in frame, it will cause a page fault, which load the page from the file or the swap slot.

I check if the page is writable before writing to it. If not, I kill the page.

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.

I use a `file_lock` for file read and write, a `vm_lock` for SPT, and a `swap_lock` for the swapping process.
It makes the code easy to read and is easy to implement, also enforces both synchronization and parallelism.