Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Hao Wang tony.wanghao@stu.pku.edu.cn

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

I passed all the test on my pc.

In this doc all the function explanations regarding struct members/variables are given in the form of comments.

> 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.

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 `splpagetable.h`:

```
/* type of supplementary page entry */
enum page_type {
    PG_FILE,
    PG_ZERO,
    PG_MISC,
    PG_SWAP
};
/* supplementary page entry, or SPE */
struct spl_pe{
    enum page_type type;    /* type of this page */
    struct file *file;      /* the file where the page is stored */
    off_t offset;           /* file offset */
    uint8_t *upage;         /* user page address */
    uint8_t *kpage;         /* physical frame address */
    uint32_t read_bytes;    /* #bytes to be read from file */
    uint32_t zero_bytes;    /* #bytes to be set to zero */
    size_t slot;            /* swap slot this page is in */
    bool writable;          /* is writable */
    bool present;           /* is present in physical memory */
    struct hash_elem elem;  /* hash elem */
};
```

In `frame.h`:

```
/* frame entry, or FE */
struct frame{
    void *frame;            /* the frame this FE represents */
    tid_t tid;              /* TID of the thread holding the frame */
    struct spl_pe *spl_pe;  /* the SPE of this frame */
    struct thread *thread;  /* the thread holding this frame */
    struct list_elem elem;  /* list elem */
    struct lock frame_lock; /* lock for frame loading */
};
```

In `frame.c`:

```
static struct list frame_table;     /* global frame table */
static struct lock ft_lock;         /* lock for frame table access */
```

In `process.h/struct process`:

```
    struct hash spl_page_table;     /* supplementary page table for this
process */
```

ALGORITHMS

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

The functions of each members of SPE is given in `A1` already.

In reality, we usually needs to find a SPE with the given user address. Since we use a hash table as the implementation of SPT, we could conveniently use `hash_find` to get the corresponding SPE. After that, we could dereference the pointer and access the data stored in the SPE.

> 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 each frame table entry, I record a pointer pointing to the supplementary page entry corresponding to the user page holding it. When the kernel needs to inspect the dirty/accessed bit, it checks the page table of the process holding the frame for the accessed/dirty bit. Thus, we only need to ensure that the preceding pointer is synchronized, which is guaranteed by `evict_lock`.

SYNCHRONIZATION

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

If both `palloc_get_page` succeeded, we use `ft_lock` to prohibit concurrent access to `frame_table`. If both of them needs eviction to get a new frame, the `evict_lock` would be acquired before getting into the eviction part, thus guaranteeing the avoidance of potential races.

RATIONALE

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

I used a `hash` (hash table) to implement the supplementary page table. I do this mainly because most of the cases where I need to access the SPT, I need to find the corresponding SPE with its user page address. Hash table is the best data structure for this since if I set the hash value according to the user page address each entry corresponds to, the search process takes almost constant time (not rigorously speaking).

I used a `list` to implement the frame table, since the main reason we have it is that we need to use the **CLOCK** algorithm, which traverses the list until the accessed bit of the frame under inspection is set to `false`. Thus, using a list is the most intuitive way in terms of data structure abstraction.

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 `frame.h`:

```
struct lock evict_lock;     /* lock to synchronize eviction */
```

In `frame.h/struct frame`:

```
    bool evictable;         /* allow pinning down */
```

In `frame.c`:

```
static struct list_elem *evict_pt;  /* pointing to the next frame ready to
                                       inspect for eviction */
```

In `swap.c`:

```
static struct block *swap_device;   /* swap device from BLOCK_SWAP */
static struct bitmap *map;          /* the bitmap recording the slot
assignment */
static struct lock swap_lock;       /* lock for swap device access */
```

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 `evict_pt` to traverse the `frame_table` in order to find a frame for eviction.

The way this works is according to the **CLOCK** algorithm. I wrote two help functions, `next_frame` and `prev_frame`, to help moving the pointer as if it were on a circular queue. Each time the obtainment of a frame requires eviction, the `evict_pt` starts from the last examined frame to check its accessed bit. If the bit is unset, this frame is the one to evict; or set the bit to `false` and move to the next frame in the queue.

The algorithm is not guaranteed to return, if the system schedules the threads so that each frame, upon inspection, is always accessed (even after the first round of examination).

> 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 the frame?

The `pagedir_clear_page` is called to clear the page out of the page table of Q and mark the user page as not present. Additionally, we can access the corresponding supplementary page entry from the frame entry and change the SPE to not present. If the page is evicted, the SPE's type would be changed to `PAGE_SWAP`, and the swap slot would be kept in the SPE.

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_device` are global structures. The `spl_pt` is exclusive for each process.

There is a big lock for `frame_table` and `swap_device` to exclude concurrent access. Additionally, each entry of frame table might be under loading when a page loading or swapping in happens. This does not necessarily change the frame table. Thus, I implemented a lock for each frame table entry. I also included an `eviction_lock` to ensure that when two evictions cannot happen concurrently.

Except a few cases, all the locks are acquired and released within the same function, which reduces the risk of deadlock. For the cases where several locks are acquired, the order of acquiring and releasing locks are specifically designed so that the deadlocks are avoided.

> 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?

We implemented a lock for each frame to protect the loading process of the frame. Basically, if the page is being evicted, P would get hold of its `frame_lock`. Additionally, P would clear this page from Q's page table, which prohibits Q from accessing or modifying the frame any more.

> 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?

`frame_lock` is designed for each frame table entry such that when the frame is being written by P, Q cannot evict or modify the frame content. Before eviction, the process needs to first get hold of the lock, which outrules the possibility of races.

> 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 manually bring in the pages using `load_page` before any system calls that might cause a page fault. Thus, any page fault during the system call indicates faulty pointers (or page loading in) and attempted accesses to invalid virtual addresses. After bringing the pages, the `evictable` flags of the frames loaded is set to `false`, which indicates that the frames are **pinned down** until the system call finishes.

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 thinks some of my design is more "hands off", like the `evict_lock`, `ft_lock`, and `swap_lock`. I tried using more small blocks but couldn't manage to escape the shadow of deadlocks. Additionally, using these two big locks does not affect the efficiency of the system to a large extent, since they are carefully designed to be put in places only necessary.

Some of the other designs falls more on the other end of the spectrum. I implemented a separate lock for each frame entry, i.e., `frame_lock`. Additionally, I put a bit in frame entry to indicate that the corresponding frame is not evictable. I think the reason here is that the loading of the frame should be seen as a process for each individual frame. Additionally, when implementing system calls, it is more efficient to pin down a few frames rather than the whole user memory.

Overall, I designed the kernel in terms of three factors: simplicity, intuition, and efficiency.