Project 3a: Virtual Memory

Preliminaries

> Fill in your name and email address.

Chang Shi 2300017795@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.

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.

1. Add a new struct `frame_table_entry` to be put in `frame_table` list, recording information about the frames.

```
 struct frame_table_entry {
    void *frame;                          /* frame address */
    struct sup_page_table_entry *spte;  /* relevant information in
supplemental page table */
    bool pinned;                          /* whether it can be swapped out */
    struct thread* thread;                /* the thread owning the frame */
    struct list_elem elem;
 };
```

2. Add a `frame_table` list as the frame table and a lock `frame_lock` to ensure correct synchoronization.

```
static struct list frame_table;
static struct lock frame_lock;
```

3. Add a pointer `ptr` to elements in the frame table, used for clock algorithm.

```
static struct list_elem* ptr = NULL;
```

4. Add a new struct `sup_page_table_entry` to be put in `sup_page_table` list, recording additional information about pages.

```
struct sup_page_table_entry{
    void* vaddr;                /* virtual address */
    bool is_loaded;             /* whether the page has been loaded */
    struct file* file;          /* source file */
    off_t file_offset;
    uint32_t read_bytes;
    uint32_t zero_bytes;
    bool writable;              /* whether the page is writable */
    struct list_elem elem;
    void* frame;                /* the frame allocated to the page */
    int slot;                   /* the swap slot index */
};
```

5. Add a `sup_page_table` list in each process as the supplemental page table.

```
struct thread{
   // something
   struct list sup_page_table;
   // something
}
```

ALGORITHMS

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

We just scan over the `sup_page_table` list of current process, and search for the entry whose `vaddr` is consistent with 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?

I don't have implemented page sharing yet, so it would never be a problem.

SYNCHRONIZATION

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

Only one of them can acquire the `frame_lock` at one time, so their requests for new frames are dealt with sequentially.

RATIONALE

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

The supplemental page table record virtual-to-physical mappings, and I directly include the mappings in the structure.

The frame table record physical-to-virtual mappings, and I include the `spte` pointer to the relevant supplemental page table entry, since other information besides the virtual address is often used or modified as well.

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.

1. Add a `swap_slot` block as swap slot, a variable `slot_count` to record the total number of original slots, a `swap_map` bitmap to record whether a slot is available and a lock `swap_lock` to ensure correct synchronization.

```
struct block* swap_slot;
block_sector_t slot_count;
struct bitmap* swap_map;
struct lock swap_lock;
```

2. Add a global definition `K` as the number of block sectors being occupied once.

```
#define K PGSIZE / BLOCK_SECTOR_SIZE
```

3. Add two global definitions `SWAP_ERROR` and `SWAP_NONE` as the special status flags.

```
#define SWAP_ERROR -1
#define SWAP_NONE -1
```

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 clock algorithm.

The following process is repeated until a frame is chosen to evict or loop continues for over 2 rounds (indicating that there are no available frames to evict).
We check the access bit of the current frame. If not accessed recently, we choose this page as the victim. Otherwise, we set the access bit to 'not accessed', and move on to the next frame.

> 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?

First, we call `pagedir_clear_page` to remove the previous mappings in process Q's page directory.

Next, we update the supplemental page table. (It's unnecessary in practice, so I don't have implemented this step.)

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 have only used two lock in my design, one for frame table management and the other for swap slot management.

Acquisition of `frame_lock` always precede that of `swap_lock`, which prevents deadlock.

> 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 first thing to do after picking a victim page is to remove the previous mapping in Q's page directory, preceding steps such as freeing the old frame and swapping the new page in. Q will get page fault when intending to access or modify the page.

In Q's page fault handler, `frame_alloc` is called. Before P finishes evicting Q's page and swapping in the new page, Q just waits for the `frame_lock` held by P. In such way we avoid a race.

> 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 allocating a new frame, the `pinned` flag is often set true to ensure that they would not by evicted. After that, we reset the flag to false.

> 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. The supplemental page table records whether the page is currently stored in the swap slot (and which blocks). We just swap in pages from target blocks.

For invalid virtual address or attempt to write to unwritable pages, we just call `exit(-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.

My design falls in almost middle of this continuum, as noted in Question B5.

This design protects global data structures well and also avoid much additional trouble with more locks.

I once intended to implement this lab with locks relevant to each frame or swap slot block, which seemed to allow for high parallelism.
However, substantial homework and TOEFL exam left me no time to think over and implement this complicated design.

Grading

All tests required for Lab3a passed in my local environment.

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
```

```
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/vm/pt-grow-stack
FAIL tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
FAIL tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
FAIL tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
FAIL tests/vm/mmap-read
FAIL tests/vm/mmap-close
```

```
pass tests/vm/mmap-unmap
FAIL tests/vm/mmap-overlap
FAIL tests/vm/mmap-twice
FAIL tests/vm/mmap-write
FAIL tests/vm/mmap-exit
FAIL tests/vm/mmap-shuffle
FAIL tests/vm/mmap-bad-fd
FAIL tests/vm/mmap-clean
FAIL tests/vm/mmap-inherit
FAIL tests/vm/mmap-misalign
FAIL tests/vm/mmap-null
FAIL tests/vm/mmap-over-code
FAIL tests/vm/mmap-over-data
FAIL tests/vm/mmap-over-stk
FAIL tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
22 of 113 tests failed.
```