CS 130 Project 3: User Programs Design Document

Tianyi Zhang
2018533074
zhangty2@shanghaitech.edu.cn

Haoran Dang
2018533259
danghr@shanghaitech.edu.cn

Derun Li
2018533152
lidr@shanghaitech.edu.cn

0 PRELIMINARIES

0.1 Preliminary Comments

No preliminary comment for this project.

0.2 References

- Pintos Guide by Stephen Tsung-Han Sher: https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5e1bb4809e4b0a78012be132/1578874001361/Sher%282016%29_Pintos_Guide

1 PAGE TABLE MANAGEMENT

1.1 Data Structures

1.1.1 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 file `vm/frame.h`

```
/* List of all frame tables allocated here */
struct list frame_table;

/* Entries of frame table */
struct frame_table_entry
{
    /* Address of the frame */
    void *frame;
    /* Corresponding SPTE */
    struct sup_page_table_entry *spte;
    /* List element */
    struct list_elem elem;
};
```

- In file `vm/frame.c`

```
/* Page initialization flag */
static bool page_table_initialized = false;
/* Lock of frame table */
static struct lock frame_table_lock;
```

1.2 Algorithms

1.2.1 In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page. Each page in SPT corresponds to a frame of kernel page, allocated by `palloc_get_page()` in user pool. When loading the page in the page fault handler, the page will be set up using `pagedir_set_page()`, and meanwhile the corresponding user virtual address will be stored in the SPT. Thus, the process can access the data in this SPT, and other operations can find this page using the user virtual address, frame address, etc.

1.2.2 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? We directly check the status of the corresponding kernel page, which should be the same as they are managed by the kernel, and an accessed/dirty kernel page means an accessed/dirty user page.

1.3 Synchronization

1.3.1 When two user processes both need a new frame at the same time, how are races avoided? A lock is set in the definition of a frame table lock. When allocating a frame and pushing it to the frame table, the global `frame_table_lock` is acquired, so the race condition can be avoided.

1.4 Rationale

1.4.1 Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings? Such a structure adds additional necessary information when not affecting the original function. This improves the stability of the SPT, and makes it convenient to implement memory mapped files and swap.

2 PAGING TO AND FROM DISK

2.1 Data Structures

2.1.1 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 file `vm/swap.c`

```
/* Store the info of swap slots */
static struct bitmap *swap_bitmap;
/* Swap block */
static struct block *swap_block;
/* Size of swap block */
static size_t swap_size;
```

- In file `vm/page.h`

```
/* States of the pages in SPTE */
enum state
{
    /* Zero page */
    ALL_ZERO = 0,
    /* Now on frame */
    ON_FRAME = 1,
    /* Now in swap */
    IN_SWAP = 2,
    /* Loaded from file, evict to file */
    FROM_FILESYS = 3,
    /* Loaded from file, evict to swap */
```

```
    FROM_FILESYS_SEGMENTS = 4
};

/* Entries of supplemental page table */
struct sup_page_table_entry
{
    /* Corresponding FTE */
    struct frame_table_entry *fte;

    /* Owner thread */
    struct thread *owner;
    /* User virtual address */
    void *user_vaddr;
    /* Writable flag */
    bool writable;
    /* Current status of the page */
    enum state status;

    /* Last access time */
    uint64_t access_time;
    /* Dirty flag */
    bool dirty;
    /* Accessed flag */
    bool accessed;

    /* Swap index if swapped out */
    size_t swap_index;

    ...

    /* List element */
    struct list_elem elem;
};
```

- In file `thread/thread.h`

```
struct list
{
    ...

    /* Supplemental page table */
    /* List of supplemental page tables */
    struct list sup_page_table;
    /* Lock of the list of SPT */
    struct lock sup_page_table_lock;

    ...
}
```

2.2 Algorithms

2.2.1 When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict. Function `find_entry_to_evict()` is called when choosing a frame to evict. In this function, the list of frame table is sorted according to the access time of the corresponding entry in SPT, and then it returns the first elem, which has the smallest access time, so this implements the LRU policy.

2.2.2 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 old frame is cleared and freed after saved into the swap or file, and the address of this frame in its corresponding SPT entry is set to NULL to avoid collision.

Then we allocate a new frame table entry with a frame here, store it in the new SPT entry, and install it into the `pagedir` of the new thread when loading the page.

2.2.3 Explain your heuristic for deciding whether a page fault for an invalid virtual address should cause the stack to be extended into the page that faulted. When a page fault occurs, we first dump the information of this fault. Then we look into the following flags:

- `not_present`: Whether the fault is due to not present page in memory, or writing to a read-only page. In the former condition, we continue to check the following conditions; in the latter condition, we terminate the program if in user context.
- `on_stack`: Whether the fault address is a valid stack address.
- `in_frame`: Whether the fault address is extending the stack in order instead of directly jumping to another place.

Only if both `on_stack` and `on_frame` are true, we extend the stack by allocating a new frame.

2.3 Synchronization

2.3.1 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.) A global lock of frame table `frame_table_lock` is set. Since all operations of the frame table shares one lock, the condition of circular wait cannot happen, and thus no deadlock here.

2.3.2 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 very first thing during the eviction of a page is clear the page by using `pagedir_clear_page()`. After this is impelemented, Q cannot access this page until it is reloaded.

Any changes on the frame table, including allocating new frames or evicting a frame, should be done with `frame_table_lock` acquired, so one cannot load back a page when a page is being evicted.

2.3.3 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 `load_page` also acquires the lock `frame_table_lock`, so the page cannot be evicted before the loading process finished.

2.3.4 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?

This problem is firstly handled in syscall level, based on our implementation in project 2. The validation of address (buffers or strings) is cheched in wrapper functions, which ensure to avoid access to page-out pages.

Later even if the access falls into the page fault handler, we will validate the access to check whether it is a valid process, and grow the stack, fetch back the evicted page or terminate the program as in user programs.

2.4 Rationale

2.4.1 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. A `frame_table_lock` is created to deal with synchronization problem, which is used in functions such as allocation and free. This lock ensures that frame table can only be modified by only one thread at one time. Since the frame table is globally accessed, it is guaranteed by this single lock.

For each page table, we still set a lock `sup_page_table_lock` for each process. This lock guarantees that the access to supplemental page table is synchronized.

3 MEMORY MAPPED FILES

3.1 Data Structures

3.1.1 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 file `threads/thread.h`

```
struct thread
{
    ...

    /* Memory-mapped files */
    /* Next identifier for mmap files */
    mapid_t next_mapid;
    /* List of memory-mapped files */
    struct list mapped_files;
};
```

- In file `userprog/syscall.h`

```
/* Map region identifier. */
typedef int mapid_t;
/* Map failed identifier. */
#define MAP_FAILED ((mapid_t) -1)

...

/* Mapped file entry point. */
struct mapid_entry
{
    /* Mapped file */
    struct file *file;
    /* Length of mapped file */
    size_t file_length;
    /* Mapped file identifier */
    mapid_t mapid;
    /* Corresponding user virtual address */
    void *user_vaddr;
    /* Whether this mapid is freed */
    bool freed;

    /* List element */
    struct list_elem elem;
};
```

- In file `vm/page.h`

```
struct sup_page_table_entry
{
    ...

    /* For memory mapped files */
    /* Mapped file */
    struct file *file;
    /* Offset in the file for this page */
    off_t file_offset;
    /* Bytes from data */
    off_t file_bytes;
    /* Bytes of extra zeros */
    off_t zero_bytes;

    ...
};
```

3.2 Algorithms

3.2.1 Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages. A page that belongs to a memory mapped file is much the same as that of an allocated memory. Both of them needs a user virtual address, a frame (i.e. corresponding kernel page), various parameters, etc. The difference is that a SPT of a mapped file stores the info of the file, while a SPT of page in swap stores the `swap_index`.

We defined a special status of SPT: `FROM_FILESYS`. Different from `IN_SWAP`, when the information of this page is stored in a file instead of the swap, it will be set into this status, so when this page need to be loaded, the `load_page()` will directly load the data in the file, instead of searching in the swap.

The special identifier of a SPT of a memory-mapped file is that it stores the information of the file, which is in the `file` pointer of the SPT structure. When this is not `NULL`, data will be stored in this file and the status will be set to `FROM_FILESYS`, but in a normal eviction process, the data are stored in the swap and the status is `IN_SWAP`.

`file_offset` and `file_bytes` are to locate the place in file of this page. By using this we can handle the situation where a file is mapped to multiple pages.

3.2.2 Explain how you determine whether a new file mapping overlaps any existing segment. When mapping a file, we go through the pages from the given `addr` to `addr+size` of the file, stepping in `PGSIZE` and checking whether the user virtuall address exists in the SPT of this thread. If that exists, that means this memory is

allocated to something else before, so we can detect that this file mapping overlaps an exsiting segement, and terminate the process due to violation.

3.3 Rationale

3.3.1 Mappings created with 'mmap' have similar semantics to those of data demand-paged from executables, except that 'mmap' mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain why your implementation either does or does not share much of the code for the two situations. They share most of the code, but although they are all lazily loaded from file, the operations to handle them after loaded from file are different, so we defined a special SPT status `FROM_FILESYS_SEGMENTS` to record this state.

The page allocation process is the same between these two situations, except for the final state. To distinguish the behavior, when loading this page for the first time, we clear the file information for SPT with status `FROM_FILESYS_SEGMENTS`, which indicates that they are segments and should be written back to swap.

Since most of the code is the same, we choose to share the code by nesting the functions.

4 SURVEY QUESTIONS

In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time? Working on the functionality of a page is obviously more harder than we thought. Sometimes the program falls into a page fault with address `NULL` but GDB `backtrace` cannot trace before `intr_entry()`, and that makes it really difficult to find the cause.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design? The memory management method of this project differs a lot from other programs, giving us an new overview of memory from the OS side. The lazy load process is the most special one, as the memory allcation does not really allocate a piece of memory in real time. Meanwhile, swap and memory-mapped files also gives us a special look into how memory cooperates with files.

Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading? Not yet.

Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Not yet.

Any other comments? Not yet.

CONTRIBUTORS

Task		Tianyi Zhang	Haoran Dang	Derun Li
Task 1 Page Table Management	Concept	✓	✓	✓
	Implementation	✓	✓	✓
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓
Task 2 Paging to and From Disk	Concept	✓		✓
	Implementation	✓		✓
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓
Task 3 Memory Mapped Files	Concept	✓	✓	✓
	Implementation		✓	
	Debugging	✓	✓	✓
	Design Document	✓	✓	✓