CS130 Project 3: Virtual Memory Design Document

Group 13

Tianyao YAN
yanty@shanghaitech.edu.cn

Wenfei YU
yuwf@shanghaitech.edu.cn

I. Introduction

This document answers the questions in the template for Project 3.

II. Page Table Management

A. Data Structures

- 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 page.h:

```
1  struct page_table_entry
2  {
3      void *physical_page;
           /* The physical address of this
            page. */
4      void *virtual_page;
           /* The virtual address of this
           page. */
5      struct hash_elem helem;
           /* Hash element. */
6      enum pte_status status;
           /* The status of this page. */
7      int swap_index;
           /* The swap index if swapped
           out. */
8      struct file *file;
           /* The file pointer. */
9      off_t ofs;
           /* The file offset. */
10     uint64_t read_bytes, zero_bytes;
           /* Readbytes and zerobytes in
           this page. */
11     bool writable;
           /* Writable or not. */
12 };
```

In thread.h:

```
1  struct thread
2  {
3      ...
4      struct hash page_table; /* Page
           table. */
5      ...
6  }
```

B. Algorithms

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

  We do it in the function page_load(page_table, pagedir, virtual_page).

  1) Find the page_table_entry based on virtual_page in page_table.
  2) If the page is already in the frame, do nothing.
  3) Allocate space for the page in the frame.
  4) Then for different statuses of this PTE, we use different ways to load the data. If ZERO, we use memset. If SWAP, we use swap_in. If MMAP, we use file_read.
  5) Call pagedir_set_page.
  6) Change the status of this PTE into FRAME and set dirty to false.

- 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 only need to know the accessed and dirty bits of the page in the frame since every page can only be accessed and modified when it's in the frame, and these bits will be got by pagedir_is_accessed and pagedir_is_dirty.

C. Synchronization

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

  When getting a new frame, we call palloc_get_page, and in this function, there is a lock to ensure that only one page can be allocated at the same time.

D. Rationale

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

  By using hash table, the speed of looking up the page_table_entry can be very fast.

III. Paging to and from Disk

A. Data Structures

- 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 frame_table_entry
{
    void *physical_page;   /* The physical address of the frame. */
    void *virtual_page;    /* The physical address of the frame. */
    struct thread *owner;  /* The thread that owns this frame. */
    struct list_elem elem; /* List element. */
};
```

In frame.c:

```
static struct list frame_table;  /* Frame table. */
static struct lock frame_lock;   /* Frame lock */
```

B. Algorithms

- When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

  We choose the page that has not been accessed before, and then set the page to be unaccessed. And if all the pages are accessed, we search for the unaccessed page again. Since the page is set to unaccessed in the first loop, so in the second loop there must be at least one unaccessed page. The code is as followed.

```
for (e = list_begin(&frame_table); e != list_end(&frame_table);
e = list_next(e))
{
    struct frame_table_entry *fte = list_entry(e, struct frame_table_entry, elem);
    if (pagedir_is_accessed(thread_current()->pagedir, fte->virtual_page))
    {
        pagedir_set_accessed(thread_current()->pagedir, fte->virtual_page, false);
        continue;
    }
    fte_evict = fte;
}
if (!fte_evict)
{
    for (e = list_begin(&frame_table); e != list_end(&frame_table);
    e = list_next(e))
    {
        struct frame_table_entry *fte = list_entry(e, struct frame_table_entry, elem);
        if (pagedir_is_accessed(thread_current()->pagedir, fte->virtual_page))
        {
            pagedir_set_accessed(thread_current()->pagedir, fte->virtual_page, false);
            continue;
        }
        fte_evict = fte;
    }
}
```

- 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?

  When process P obtains a frame by calling frame_get_page, it will call frame_table_entry_set if the page successfully allocated. And in that function, the owner of the frame will be set to thread_current(), which means process P here.

- 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.

  If fault_addr >= (PHYS_BASE - STACK_SIZE) && (fault_addr >= f->esp || fault_addr == (f->esp - 4) || fault_addr == (f->esp - 32)) , we choose to grow the stack.

C. Synchronization

- 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.)

  In VM, there's only one lock which is frame_lock. We prevent deadlock by ensuring that the thread will not acquring any resource that cannot be allocate immediately when holding this lock.

- 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?

When process P is evicting a page, the frame_lock is held by it and process Q cannot do anything on the frame since it cannot acquire frame_lock now and can only wait for process P finishing its eviction.

- 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?

  Process P and Q are in different threads, so Q will not modify the page owned by P. Only eviction is needed to be considered. However, in our eviction strategy, it will only evict the page unaccessed or the first page in frame_table if every page is accessed. So when reading in, the page is accessed and this page is at the end of the frame_list. So Q will not evict the page.

- 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?

  We use page faults to bring in pages. The paged-out pages are in status SWAP, and we only need to do swap_in to load it back.

D. Rationale

- 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.

  We choose to use frame_lock which is a single lock since it is easy to implement and easy to prevent deadlock.

IV. Memory Mapped Files

A. Data Structures

- 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 page.h:

```
typedef int mmapid_t;

struct mmap_entry
{
    mmapid_t id;            /* MMAP
       id. */
    struct file *file;      /* File
       pointer. */
    void *va;               /* The
       mapped virtual address. */
    struct list_elem elem; /* List
       element. */
};
```

  In thread.h:

```
struct thread
{
    ...
    struct list mmap_list; /*
       Memory map list. */
    ...
}
```

B. Algorithms

- 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.

  Every thread has its own mmap_list to store the mmap_entry which has the information of the memory mapped files. In mmap, we record the information in the mmap_entry and a page_table_entry whose status is MMAP. If it is loaded, the file will be read into the frame, and the status of that PTE is changed into FRAME. And then if it is evicted, the status will change into SWAP. When munmap, if the status is FRAME or SWAP, the page may be dirty. If it is dirty, file_write is called. And if the status is still MMAP, it means the page is not loaded. So it is impossible that the page is modified and requires writeback.

- Explain how you determine whether a new file mapping overlaps any existing segment.

  By the following code:

```
for (int ofs = 0; ofs < size; ofs
   += PGSIZE)
{
    if (page_table_entry_search(&
       thread_current()->
       page_table, data + ofs))
    return -1;
}
```

C. Rationale

- 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.

  We share the part of lazy load since it makes the implement more simple.

V. Survey Questions

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want–these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

- 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?

  The task, memory mapped files, is very good. However, the other two tasks are so complicated that they take too much time and we still cannot know why and where some of the testcase always exit(-1).

- Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

  Maybe.

- 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?

  Students need to be told which functions need to be modified or added.

- Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

  For the physical and mental health of students, TA had better provide more tips.

- Any other comments?

  No.

References

- https://github.com/youcunhan/cs130-OperatingSystem-pintos
- https://github.com/Pst2000/CS130-Operating-System