Design document (Project 3: Virtual Memories)

Team 3

- ChaYoung You chacha@kaist.ac.kr
- HyunJin Jung jhjnav7@kaist.ac.kr

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.

- `threads/thread.h`
  - `struct thread`
    - `struct hash page_table` : Supplemental page table.
- `vm/frame.h`
  - `struct frame`
    - `struct thread *thread` : Thread.
    - `void *addr` : Kernel virtual address.
    - `void *upage` : User virtual address.
    - `struct list_elem elem` : List element.
- `vm/frame.c`
  - `static struct list frame_table` : Frame table.
  - `static struct lock frame_lock` : The lock for frame table.
- `vm/page.h`
  - `struct page`
    - `void *addr` : Virtual address.
    - `struct file *file` : Loaded file.
    - `off_t file_ofs` : Offset of the file.
    - `uint32_t file_read_bytes` : Number of read bytes from file.
    - `bool file_writable` : File is writable.
    - `struct hash_elem hash_elem` : Hash table element.

Algorithms

> A2: In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.

When `VM` is defined, which is able to check by `#ifdef VM`, `palloc_get_page()` calls can be replaced with `frame_alloc()` calls in `load_segment()` or stack growthing in `page_fault()`. We know the user virtual address for each frame, so `frame_alloc()` receives them and map it to the kernel virtual address of the new frame.

We can find the frame with the user virtual address of the page. `frame_table` holds every frame and each frame holds the user virtual address where it maps from. So iterating `frame_table` would gives us the frame.

> 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?

We use `pagedir_is_accessed()` and `pagedir_is_dirty()` things only, so we uses user virtual addresses only.

Synchronization

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

Every operation modifying `frame_table` is atomic by `frame_lock`. When a thread acquires a new frame, it calls `frame_alloc()` and then any other thread can't call `frame_alloc()` until that call finishes.

Rationale

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

`struct frame` has `void *addr` for the kernel virtual address, and `void *upage` for the user virtual address. Also every `struct frame` is stored in `static struct list frame_table`, so we can know the mappings.

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.

- `vm/page.h`
  - `struct page`
    - `bool valid`: Frame is not swapped out.
    - `size_t swap_idx`: Swap index of the frame.
- `vm/swap.c`
  - `static struct bitmap *swap_table`: Swap table.
  - `static struct lock swap_lock`: The lock for swap table.

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.

At first, we iterate every frame of `frame_table` until we find the frame to evict. For each frame, we check if it's accessed by `pagedir_is_accessed()`. If it's accessed, then we set its accessed bit to false and then iterate to next item. If its accessed bit is not set, then select it to evict. This is second chance algorithm.

> 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?

When a frame is evicted, then we remove the mappings from user virtual address to the kernel virtual address by `pagedir_clear_page()`. So the process Q can't retrieve the previous frame now on.

> B4: 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.

At first check whether the page is swapped out or not. If it is not swapped out, then we check that this request cause the stack growth. Ensure that `fault_addr` is user virtual address and it's higher than or equal to `f->esp - 32`, for the `PUSHA` instruction.

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.)

Make instructions modifying page table and frame table atomic by `frame_lock`. If a frame is added, then corresponding page will be also added, and we make them atomic.

> 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?

Evicting a frame also be done in `frame_alloc()`, and this is done under the `frame_lock` acquired. So when P gets a new frame, a frame of Q will be evicted and no other instructions related to `frame_table` are mutually excluded. Also the `swap_in()` can be started when a process acquired the `frame_lock`.

> 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 checking the page contains `fault_addr`, we acquire the `frame_lock`, and then release it after the processing `page_load_swap()` or `page_load_file()` things.

> 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?

We use page faults as in user programs. For invalid virtual addresses, we check if there is a page with the address. If there is no such page or it's already loaded, then exit with -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.

We use `frame_lock` only. It's hard to prevent bugs caused by multiple threads.

Memory mapped files

Data structures

> C1: 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.

- `vm/page.h`
  - `struct page`
    - `bool loaded`: Page is loaded.
    - `mapid_t mapid`: Mapping identifier.
    - `struct list_elem elem`: List element.

Algorithms

> C2: 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.

When the page fault occurs with the memory mapped page, check if it's loaded already. Since memory mapped pages are loaded lazily, we should check it. When we evict a memory mapped page, we don't use swap disk. If the dirty bit is set, we write the content of the page to the file directly.

> C3: Explain how you determine whether a new file mapping overlaps any existing segment.

In `sys_mmap()`, at first we insert pages for the file mapping. We use page-aligned addresses only, so we can check if the page with the specific address already exist or not.

Rationale

> C4: 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 you’re your implementation either does or does not share much of the code for the two situations.

We implemented them separately, since we should handle exceptions like overlappings which involves reverting the `page_insert` things.

Survey questions

> In your opinion, was this assignment or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?
>
> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?
>
> 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?
>
> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?
>
> Any other comments?