---- TEAM ----

>> Team name.

Team 19

>> Fill in the names, email addresses and contributions of your team members.

Hyeonwoo Kang <kanglib@kaist.ac.kr> (70)

Young-geol Jo <rangewing@kaist.ac.kr> (30)

>> Specify how many tokens your team will use.

1 token for Project 3-1

3 tokens for Project 3-2

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.

```
struct thread {
    ...
    struct hash page_table;
    struct hash mmap_table;
    mapid_t mmap_n;
    ...
};
```

- page_table: A supplemental page table.
- mmap_table: A map region table used by mmap system call.
- mmap_n: Stores the map region ID to be used next.

```
enum page_status {
  PAGE_PRESENT, /* Present in memory. */
  PAGE_SWAPPED, /* Swapped into disk. */
  PAGE_LOADING  /* Loading file. */
};
```

- enum page_status: Possible states of a page in supplemental page tables.

```
struct page {
  enum page_status status;
  void *vaddr;
  union _mapping {
    void *frame;
    slot_t slot;
  } mapping;
  struct _load_info {
    struct file *file;
    off_t offset;
    uint32_t bytes;
  } load_info;
  bool is_writable;
  struct hash_elem elem;
};
```

- struct page: A supplemental page table entry.
  - status: The state of the page.
  - vaddr: The virtual address of the page.

- frame: The mapped frame. (if status is PAGE_PRESENT)
- slot: The mapped swap slot number. (if status is PAGE_SWAPPED)
- file: The file to load in the page fault handler. (if status is PAGE_LOADING)
- offset: The offset in file.
- bytes: The number of bytes to load.
- is_writable: A flag to determine whether the page is read-only or writable.
- elem: A hash table element.

```
struct lock page_global_lock;
```

- page_global_lock: A global lock used for synchronization.

```
enum frame_status {
  FRAME_FREE,
  FRAME_USED
};
```

- enum frame_status: Possible states of a frame in the frame table.

```
struct frame {
  enum frame_status status;
  uintptr_t paddr;
  struct page *page;
  uint32_t *pagedir;
  struct hash_elem elem;
};
```

- struct frame: A frame table entry.
  - status: The state of the frame.
  - paddr: The physical address of the frame.
  - page: The page which the frame is mapped to.
  - pagedir: The page directory which has the frame.
  - elem: A hash table element.

```
static struct hash frame_table;
static struct hash_iterator frame_table_iter;
static struct lock frame_table_lock;
static size_t frame_count;
```

- frame_table: The frame table.
- frame_table_iter: An iterator of the frame table.
- frame_table_lock: A lock used to synchronize the frame table.
- frame_count: The number of all user frames that the system has.

---- ALGORITHMS ----

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

First, call page_lookup() with the virtual address of the given page. It looks up the supplemental page table of the current process, and returns the corresponding supplemental page table entry if the page is mapped. If the state of the entry is PAGE_PRESENT, the page is mapped to a frame that contains the user data and we can read mapping.frame field to locate the desired frame. Otherwise, the given page is not mapped to a frame due to swapping or lazy-loading.

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?

Accessing user pages only with user virtual addresses in the kernel avoids the issue. The only exceptional case is the swapping code; it performs disk operations with kernel virtual addresses because the corresponding user virtual addresses are not available at that time.

---- SYNCHRONIZATION ----

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

In our implementation, user processes have to call frame_alloc() to get a new frame. It is one of the functions that require access to the (global) frame table, and there is a lock (frame_table_lock) to ensure mutual exclusion between those functions. Therefore two concurrent calls to frame_alloc() are processed sequentially and races are avoided.

---- RATIONALE ----

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

Pintos' page tables are hardware-dependent and have very limited use. We implemented "hashed" supplemental page tables and a frame table, for the following reasons:

1. Both are very frequently accessed by the system. Looking up a hash table is pretty fast.
2. Insertion and deletion on supplemental page tables are also very frequent. List structure is also good but exhibits low search performance.

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.

```
typedef size_t slot_t;
```

- slot_t: The type for swap slot numbers.

```
static struct disk *swap_disk;
static struct bitmap *swap_table;
static struct lock swap_lock;
```

- swap_disk: The swap disk object.
- swap_table: The swap table.
- swap_lock: A lock used to synchronize the 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.

Our code implements clock algorithm. It loops through the frame table infinitely to:

1. Check if the frame is mapped to a user page.
2. Check the accessed bit of the frame to find a victim.
3. Clear the access bit of the frame.

When a victim is found, it exits the infinite loop immediately.

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?

1. Get the frame table entry of the frame.
2. Update the state of the supplemental page table entry that is mapped to the frame.
3. Remove the frame from Q's page table using pagedir_clear_page().

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.

If the page is not found on the supplemental page table and the faulted address is not over 32 bytes (IA-32 PUSHA) below the stack pointer, it will cause the stack to be extended, only within the 8 MB user stack limit.

---- 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 the swap table are global and have locks for internal synchronization. There is also a lock to synchronize between the frame eviction code and the page fault handler. To prevent deadlock between that lock and the file system lock, we placed the lock_acquire() calls carefully to avoid circular wait condition.

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?

To ensure that Q cannot access the page, the frame eviction code updates the state of the page and remove it from Q's page table. Acquiring the global lock in both processes avoids 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?

The page fault handler and the frame eviction code both acquire the global lock to avoid a race.

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 first check whether all the pages are mapped in the supplemental page table to prevent accesses to invalid virtual addresses. Paged-out pages are brought by page faults when they are accessed first time.

---- 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.

The global tables have objects for internal synchronization, but most VM code use a single global lock. Therefore it permits limited parallelism. We chose this design because proving correctness of the VM system with many locks was very hard.

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.

```
typedef int mapid_t;
```

- mapid_t: The type for map regions.

```
struct mmap {
  mapid_t mapid;
  struct file *file;
  void *start;
  void *end;
  struct hash_elem elem;
};
```

- mapid: The map region ID.
- file: The memory-mapped file.
- start: The starting address of the region.
- end: The ending address of the region.
- elem: A hash table 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.

A supplemental page table entry has different states for swap and file-mapping pages, and can store information which are needed when loading the memory mapped file. The page fault handler and the eviction process perform file I/O instead of swap operations when the page is a

file-mapping page. The eviction process writes the content of the page to the file only when the page is dirty.

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

If any page in the given region is already in the supplemental page table, page_map() will fail. Then we know that the mapping overlaps some existing segment.

---- 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.

Our mmap implementation uses same fields of a supplemental page table entry, and same method to load and write back, with demand-paged executables. Therefore they share much of the implementation details.

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 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?