---- TEAM ----

**>> Team name.**

Group 2

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

Dauren Baitursyn <biddy.as.diddy@kaist.ac.kr> (50)

Kate Abileva <k.abileva@kaist.ac.kr> (50)

contribution1 + contribution2 = 100

**>> Specify how many tokens your team will use.**

3 (1 for part 3-1, 2 for part 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.**

/\* SPT entry. \*/

struct spte

{

void \*upage; /\* Virtual address of process page. \*/

enum palloc\_flags flags; /\* Flags, associated with page allocation. \*/

enum spte\_flags status; /\* Flags that explicitly indicates page status

like if it is writable, or swapped. \*/

struct frame\_entry \*fe; /\* Frame entry to which SPTE is linked to,

if any. \*/

size\_t swap\_idx; /\* If current SPTE has been swapped, the

index of swap bitmap. \*/

struct file \*file; /\* File that that is associated with page,

if any. \*/

off\_t ofs; /\* Offset in file. \*/

uint32\_t read\_bytes; /\* Read bytes in file. \*/

uint32\_t zero\_bytes; /\* Zero bytes in file. \*/

int mmap\_id; /\* MMAP ID for current SPTE, if it allocated

as SPTE for MMAP. \*/

struct list\_elem l\_elem; /\* List element to manipulate the list of

current mapped SPT entry. \*/

struct hash\_elem h\_elem; /\* Hash element to manipulate hash SPT. \*/

};

enum spte\_flags

{

WRITABLE = 0x1, /\* Indicates if current can be written upon. \*/

SWAP = 0x2, /\* Indicates if this page is swapped. \*/

FILE = 0x4, /\* Indicates if this page is associated with file. \*/

PINNED = 0x8 /\* Indicates if this page is pinned. Used to avoid

race condition in allocating frame while loading

page in load page function. \*/

};

---- ALGORITHMS ----

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

First, the pointer will be processed. It will be checked for validity - if it is from user virtual address space. It will matched to nearest memory page aligned address if necessary.

Second, the entry from Supplemental Page Table is loaded. If the frame associated with it already in the memory, then it just returns that. If not, then it is either in the swap, or in Memory Mapped Page. In the first case, we simply load it from the swap. Otherwise, we load MMP from the file directly.

The frame allocation for SPTE is as follows. First, it will try to get the page from user pool. If there are no available free pages, then frame eviction will be conducted. It will simply look through frames that are on the memory, and using the second-chance clock algorithm.

**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?**

The pointers will be checked for legal access for user processes. If the page that address belongs to is not loaded, it will be loaded from swap area or from file. Basically, only user virtual addresses are processed. Kernel virtual address is never accessed, or at least, during exception, or syscall, we check the validity of the pointer by checking if it belong to user virtual user address since all pages that are allocated for user process are allocated from user memory pool.

---- SYNCHRONIZATION ----

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

The lock is used. Since our frame allocating and evicting functions are available for all processes, the global lock variable is declared and initialized during thread system initialization. So whenever one process tries to allocate a frame, it acquires lock blocking other processes from making race condition.

---- RATIONALE ----

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

The supplemental page table helps to store additional information related to page. Be it whether it is from swap area, or it is allocated for memory-mapped file. It provides good abstraction for the page allocation procedure. Frame table is quite flexible - the frame tables are allocated whenever they are needed. So at first we might only few frames dedicated for the initial user process. As process asks for more pages like for MMP or stack growth, it will allocate additional frames.

Also the flags for the SPT entry were declared for easy manipulation with the page installation. Through these flags the status of the page can be learned, if it is for swap, or file, writable, etc.

Hash was used to keep the pages for specific processes. The hashing is based on the user virtual page address. It allows fast retrieval and storage as well for manipulating pages. “unsigned hash\_bytes (const void \*, size\_t)” function was used to get the hash code of specific page.

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.**

/\* List entry for frame. \*/

struct frame\_entry

{

struct spte \*spte; /\* Corresponding SPT entry that is linked to

this frame entry. \*/

void \*kpage; /\* Physical address of memory that is linked

to current frame. \*/

struct thread \*thread; /\* Thread to which this frame is allocated. \*/

struct list\_elem elem; /\* List elem for the frame table list. \*/

};

#define SECTORS\_PER\_PAGE (PGSIZE / BLOCK\_SECTOR\_SIZE)

#define MAX\_STACK\_SIZE (1 << 23)

#define LOADED BITMAP\_ERROR

#define NOT\_LOADED ~LOADED

---- 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.**

The algorithm chosen for eviction is second-chance clock. It basically goes in loop looking for the frame to be evicted. If the accessed bit is set to one, then set it to 0 then go for the next frame to repeat the procedure. If frame currently being observed has been accessed, it is chosen to eviction frame. It is simply will be written to swap area if the status indicates so. Afterwards we simply discard this page by clearing the page, and freeing the page so that we have one page free in the user pool.

**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?**

Whenever process P creates a STP entry, it will also save the flags for allocating the frame when it was created. So that when allocated frame that has been used by other process previously, it will processed as needed according to flags. For instance, if the flag says that it should be all-zero page, then it will use that flag when allocating the frame.

**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.**

The page will be allocated when there is no SPTE for accessed address. Then the address is checked if it below the stack pointer for 32 bytes, since PUSHA operation could be accessing at most 32 bytes below the stack pointer. If so, then allocate a page for stack. If the pointer goes above the max stack size which is 8 MB then we terminate it. We check by subtracting the phys base address from fault address and comparing it to max stack size, which is 8 MB.

---- 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.)**

There is flag that is dedicated for that. Whenever one process asks for the specific page to be loaded, it pins it, so that during eviction procedure in other procedure, they can’t use this page as a eviction page.

**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?**

First, the lock is used in the frame allocation procedure, so that only one procedure can be involved in frame allocation. If is the case when page is being loaded, and other process tries to evict that page, then the pin flag rescues the situation. While choosing the frame to evict, the SPTEs that are flagged pinned will stay untouched. This way we can avoid race condition. Then When Q process tries to access the page that has been evicted, it will cause page fault and load another frame for that page using the pages swap\_idx attribute.

**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?**

As has been explained previously, the page is flagged as pinned when being loaded. When other processes choose potential pages to be evicted, they will miss those pages that are flagged as pinned.

**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?**

It was decided to use page faults to bring back the pages that are out. First, it is checked whether the pointer is valid by checking if it is from user virtual address space, and bigger than the start of the code segment. If the page exists, it simply brings it into the memory. Otherwise it will call exit with error code.

---- 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.**

To avoid complexities, it was decided to stick with the single lock for frame eviction, and other one for swap. This might not be good in terms of parallelism, but it provides good security over deadlock and race condition.

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.**

/\* Function that is called when SYS\_MMAP invoked. Creates a SPT

entry for the file of file descriptor fd. \*/

static int sys\_mmap (int, void \*);

/\* Function that is called when SYS\_MMAP invoked. Creates a SPT

entry for the file of file descriptor fd. \*/

static void sys\_munmap (int);

/\* Function for unmapping the file mapped page(s). If the argument is

-1 then unmaps all the file mapped pages. \*/

void page\_unmap (int);

---- 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.**

The memory mapped files are implemented with lazy loading, which is similar to executables files. The file is read when a page fault occurs. It’s also loaded into the physical memory in a similar manner.

The difference is in eviction and unmap. On unmapping, the thread iterates through the whole list of mmaps, writes dirty pages back to disk and frees the memory when necessary. When a process is exiting memory mmapped files are being unmaped.

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

Very simple procedure used to check if the mapping is being done on valid page - the page with the mapping address is searched and if that page is there, then it is simply blocked. This way it is made sure that not only existing segments are written, but also any other page that is existent.

---- 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.**

MMAP mappings are similar. But since the executables can’t be changed, we simply flag it as swap pages. Then for MMAP pages, we store all the details just like in executables - page read bytes, page zero bytes and offset, and whenever we unmap those files, we check for its dirty bit - if it has been modified, if so then we save it. Otherwise we simply discard them.

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?**