```
        +------------------------------------+
        |              CS 301                |
        | PROJECT 3: VIRTUAL MEMORY - PAGING |
        |           DESIGN DOCUMENT          |
        +------------------------------------+
```

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Eesha Kulkarni <eesha.kulkarni@iitgn.ac.in>
Rishab Jain <rishab.jain@iitgn.ac.in>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

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
  {

    /* Owned by userprog/process.c. */
    uint32_t *pagedir;              /* Page directory. */
    struct hash *pages;             /* Page table. */
    struct file *bin_file;          /* The binary executable. */

  };
struct frame
  {
    struct lock lock;       /* Prevent simultaneous access. */
    void *base;             /* Kernel virtual base address. */
```

```
        struct page *page;      /* Mapped process page, if any. */
 };
struct page
 {
        /* Immutable members. */
        void *addr;             /* User virtual address. */
        bool read_only;         /* Read-only page? */
        struct thread *thread;          /* Owning thread. */


        /* Set only in owning process context with frame->frame_lock held.
        Cleared only with scan_lock and frame->frame_lock held. */
        struct frame *frame;            /* Page frame. */


 };
```

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data
>> stored in the SPT about a given page.

The page variable is used to map frame to page, whereas the base variable has a pointer to the data in the frame, which in turn uses the kernel virtual address to get to the data.

A function in process.c is used to check the array of frames to find a frame that is not mapped to a page. If such a frame is found, it is assigned to the page that comes in. In case there is no free frame, a frame has to be evicted from the array of frames. If mapping is not achieved even after eviction, another function (try frame alloc and alloc can be attempted in the already used function - try frame alloc and lock.

>> 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 user virtual addresses are stored in the page and the kernel virtual addresses are stored in the frame. The pages and frames are connected with the help of the struct. For mapping the virtual addresses, there is also a function in process.c is used.

---- SYNCHRONIZATION ----

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

To avoid races when two user processes need a new frame simultaneously, a lock must be allocated to every frame. By getting the lock, the page enters the critical section, where synchronization is handled, the new frame is allocated and the lock is then removed.

---- RATIONALE ----

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

The simple and easy data structures for the frames and pages helps keep the implementation easy by allowing to check if a frame is allotted or being evicted or if a page has been loaded or not.

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.

```
struct page
 {

        /* Memory-mapped file information, protected by frame->frame_lock. */
        bool private;           /* False to write back to file,
                                   true to write back to swap. */
        struct file *file;      /* File. */
        off_t file_offset;      /* Offset in file. */
        off_t file_bytes;       /* Bytes to read/write, 1...PGSIZE. */
 };
```

---- 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.

If there are no free frames, the frame is evicted by using the LRU algorithm with the help of the function try alloc and lock. Maintaining track of our place in the list, we traverse twice through the list of frames. We check for the recent access of the page. If it was accessed recently, it has been used and thus, we should keep going forward till we reach the least used one. In cases

where there has been no recent access we use page out to evict a frame. The code returns NULL to if it can remove the frame using a page out.

>> 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?

The page variable in the frame is given a value NULL and is freed from the frame. Thus, by the time P obtains a frame that was earlier used by Q, the frame used by Q is freed.

---- 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.)

For synchronization purposes, a lock has been assigned in the frame struct. The locks being used are internal locks. In our code, we have ensured that the previous lock is removed before it is acquired by some other process, which means we have avoided a deadlock condition.

In case of the file system, functions involving file system operation have been used to ensure that multiple process don't try to acquire the locks at the same time, to avoid the deadlock conditions.

>> 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?

The lock assigned in the struct - frame is used here. To avoid a race between P evicting Q's frame and Q faulting the page back in, the lock needs to be acquired before the page is evicted. Thus, when Q comes across a fault, P would be blocked till the eviction process is complete

>> 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 concept of frame marking was used here. During loading of frames, they are checked for marking. If it turns out that the frame is marked, it cannot be used and thus, cannot be given to

any other process Q before being unmarked. Unmarking would occur once the loading is complete.

>> 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?

The user data is accessed from the virtual address by the kernel and the pages are brought in with the help of page faults. The page faults come into picture since the user data gets checked before being accessed. Firstly, the pages are checked, followed by the checking of corresponding frames to see if they can be evicted or not. Based on the eviction process, the frame is evicted.
In the page checking step, the invalid virtual addresses can be detected and are then rejected. If the access itself is invalid, then the process will have to exit.

---- 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.

In our implementation, the frame struct has been assigned only one lock and the use of locks is only for the structures that require protection and are not held for long periods of time, continuously. This method ensured we didn’t face or minimized the deadlocks. The aim was to keep the use of locks such that concurrency is maximum.
However, to avoid the race conditions for two processes trying to evict the same frame, the code had to be slightly changed. The use of local locks, used only when required, helped with this part without hampering the parallelism, but a possibility of deadlock came into picture, but it was eventually solved by trying to remove circular locks, if any.