+--------------------------------+
| CS 301 |
| PROJECT 4: VIRTUAL MEMORY - MMAP |
| 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.

STACK GROWTH
==========

---- 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.

No new struct has been added or changed.

---- ALGORITHMS ----

>> A2: 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.

When a page fault occurs, we need to first check if the user program is more than the size of the stack. The size of the stack is measured by taking the difference between STACK_MAX and PHYS_BASE. If the size is greater, a new page needs to be added and the stack grows.

It is necessary to make sure that the stack pointer is exactly below the stack, before the stack starts growing. For the 32 bytes to be pushed together, it is essential that the address of the fault lies within 32 bytes of stack pointer.

MEMORY MAPPED FILES

---- 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 mapping
 {
   struct list_elem elem;      /* List element. */
   int handle;                 /* Mapping id. */
   struct file *file;          /* File. */
   uint8_t *base;              /* Start of memory mapping. */
   size_t page_cnt;            /* Number of pages mapped. */
 };
```

---- ALGORITHMS ----

>> B2: 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.

For the memory mapped files to integrate in the virtual memory subsystem, the struct mapping mentioned above is used. This struct maps the threads containing file references from the virtual memory to the main memory and are then stored in the main memory.

For a page fault, we have to first find the page that is to be evicted. For a private page, during the eviction process, we cannot write back to the original file. Thus, it is swapped to the swapping space. If it is not a private page, simply writing back to the file works just fine. Both these activities happen for dirty pages. For clean pages, with no data being changed, just freeing the page does the work.

>> B3: Explain how you determine whether a new file mapping overlaps
>> any existing segment.

Pages are mapped one after the other from a file to the necessary address. If the hash table finds any duplicate file there is an overlap of the new file with an existing segment. For every new file, the previous mappings get unmapped.

---- RATIONALE ----

>> B4: 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.

In our code, if any page is changed, then the implementation writes pages back to the original file. Demand paging involves lazy loading such that only the required pages are taken into the physical memory.