+-------------------------+

| OS 211 |

| TASK 3: VIRTUAL MEMORY |

| DESIGN DOCUMENT |

+-------------------------+

---- GROUP ----

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

Huanyao Rong hr1316@ic.ac.uk

Pinchu Ye py416@ic.ac.uk

Xi Jiang xj616@ic.ac.uk

Chuanqing Lu cl5616@ic.ac.uk

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, or notes for the
>> markers, 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/FRAME MANAGEMENT

==========

---- DATA STRUCTURES ----

>> A1: (2 marks)
>> Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or
>> enumeration that relates to your supplemental page table and
>> frame table. Identify the purpose of each in roughly 25 words.

For supplemental page table:

```
enum page_src
{
  SRC_RAM, //Mark the userpage that have been mapped to respective kernel page and frame.
  SRC_LOAD //Mark the userpage that would load file lazily in page fault.
};

struct load_info
{
  struct file* file; //The file that would be load in page fault.
  rom_id_t rom_id; //id of the read only memory user page.
  off_t start_read; //The position where the the file will be read at.
  size_t page_read_bytes; //How many bytes the file will be read.
};
```

//union is used that page table element "_sup_pt_elem" is able to save different infomation based on the enum "page_src" it has with less memory used,
since no page table element would have two different infomation at the same time.

```
union sup_page_info
{
  ...
    void* kpage;          // The kernel page that the user page is mapped to.
                  This is only for SRC_RAM, cannot be used by SRC_M_FILE.
  ...
```

```
    struct load_info load_info;    // The loading information that is needed for lazy
                                                    loading elf during page fault. This is
for SRC_LOAD.
    };

    typedef struct _sup_pt_elem
    {
      const void* userpage_addr; // The userpage address.
      struct hash_elem elem; // The elem that is used for hash table
"sup_page_table".
      enum page_src src; // This enum marks different page table elem to perform
                                    differently in different switch cases in many functions
such as                                page_fault().
      union sup_page_info u; // The information saved to the respect of page_src
enum
      uint32_t* pt_entry; // The entry to the kernel page, which keeps tracking
                                the presence of the page.
    }sup_pt_elem;


    struct thread
    {
     ...
     struct hash sup_page_table; /* Every thread has its own supplemental page
table
                                as they use independent user address space.
     ...
    }
```

For frame table:

```
typedef struct _frame_table_entry {
   int64_t freshness; //record the timer ticks since last access or modification.
                                 It will be reset to 0 after then.
  struct list u_page_sup; // The list for page table elements which stores the
                                       user pages that map to this frame.
   bool is_allocated;      // True if the frame is reserved. i.e. returned by palloc.
   bool is_pinned;         // True if the frame is not eligible for eviction.
   rom_id_t rom_id;        // Read only memory id, 0 if not ROM, positive if ROM.
}frame_table_entry;


// An array for frame_table_entries.
typedef struct _frame_table
{
   frame_table_entry* frame_tab;    //The head frame_entry of frame table.
   size_t num_of_frame;           //The total number of frame_entries.
   uint8_t* user_page_start;       //Start of user address space.
}frame_table_info;

frame_table_info g_frame_table; //Global frame table.
```

---- ALGORITHMS ----

>> A2: (2 marks)

>> Describe your code for locating the frame, if any, that contains

>> the data of a given page.

In frame.c, find_frame_entry(), with the input of a pointer to kernel address of a user page, returns a frame_table_entry pointer which contains all needed information for a frame. The function computes the user page index through finding how many page sizes are away from the user start address. The user index is then the same as the respective frame index in the frame table.

>> A3: (2 marks)

>> How have you implemented sharing of read only pages?

We have maintained a global data structure that maps executable file name and user virtual address of a specific page to a rom id. If the page is writable, rom id is 0; if the page is ROM, rom id is a positive number.

Such rom id would be stored in frame table element and suplementary page table element when suplementary page table is to be loaded or to be swapped. When a page is to be loaded from swap or from file system, we will check if the rom id is already in the loaded frame. If it is already loaded, install that page to user memory. If not, load as usual and set rom id in frame table entry as the rom id stored in the suplementary frame table entry.

The rom id is allocated when segment is loaded in load segment if this ROM page is not loaded already.

---- SYNCHRONIZATION ----

>> A4: (2 marks)

>> When two user processes both need a new frame at the same time,

>> how are races avoided? You should consider both when there are

>> and are not free frames available in memory.

In palloc_get_multiple(), the method bitmap_scan_and_flip() is used to reserve a kernel page. Since our design has a one-to-one mapping from kernel address to frame address. Obtaining a new kernel page is the same as getting a free frame. Race conditions are avoided by locking the pool lock before calling the bitmap method above. If no free frames are available, swap method would be called to release needed free frame space, which is also synchronized with lock “ft_spt_lock”.

---- RATIONALE ----

>> A5: (2 marks)

>> Why did you choose the data structure(s) that you did for

>> representing the page table and frame table?

We use hash structure for page table and array for frame table. For page table, using hash allows us to have average O(1) time complexity when doing search and insert. For frame table, we store an array in the free memory above 1MB. The reason why I use an array is that array can have O(1) complexity to edit any of the element. However, if we use hash for frame table, we must add a lock when accessing the hash table, but we cannot do so in thread_yield or any other function being called when switching threads.

Array may waste some space, but it is not so much. And if all of the memory is used, no space is wasted for array frame table.

PAGING TO AND FROM DISK

==========

---- DATA STRUCTURES ----

>> B1: (1 mark)
>> Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration that relates to your swap table.
>> Identify the purpose of each in roughly 25 words.

```
enum page_src
{
  SRC_SWAP,
  ...
};
```

SRC_SWAP to indicate that the page is expected to be found in swap block.

```
union sup_page_info
```

```
{
  swap_info swap_info;        // for SRC_SWAP.
  ...
};
```

swap_info is the structure that contains supplemental information for user page stored in swap device.

```
typedef struct _swap_info
{
  swap_idx_t swap_idx;
  rom_id_t rom_id;
}swap_info;
```

swap_idx is the index used to locate page in swap device.

rom_id used to identify read only pages, same rom_id indicates data stored on pages are identical.

```
typedef uint32_t swap_idx_t;
```

swap_idx_t being type of uint32_t to be consistent with block_size() return value.

```
swap_idx_t swap_size;
```

This is the total number of pages that could be stored into swap block.

```
struct block* swap_device;
```

This is the actual device used to store pages evicted.

```
struct bitmap* used_map;
```

A bitmap to indicates which sectors are in use within the swap block.

---- ALGORITHMS ----

>> B2: (2 marks)
>> When a frame is required but none is free, some frame must be
>> evicted. Describe your code for choosing a frame to evict.

When there is not any free frame, the function find_page_to_evict() traverse all the allocated, installed, unpinned frames to find the one whose installed user page has smallest "freshness" value, that is, the least recent touched page.

>> B3: (2 marks)
>> 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 there is a swap between pages, the “sup_pt_elem”s mapped to the chosen frame for eviction all have SRC changed to SRC_SWAP and idx set to swap_idx, The "frame_table_entry" then clears the “u_page_sup” it holds in order to take in new “sup_pt_elem”s. “sup_pt_elem” swapped into the frame has it’s SRC changed to SRC_RAM and push itself to the back of the “u_page_sup” list held by the frame. After all these steps. the supplementary page table and frame table now indicates that P owns the frame previously used by Q and Q is in swap device.

---- SYNCHRONIZATION ----

>> B4: (2 marks)
>> Explain how your VM synchronization design prevents deadlock.
>> (You may want to refer to the necessary conditions for deadlock.)

There is only one lock for frame table and supplementary page table, therefore there is no chance that deadlock would occur, deadlock could only occur when there are two or more locks.

>> B5: (2 marks)
>> 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?

The eviction process will clear the present bit of the page entry in the first place, to make sure that rest of the operation finishes before the evicting page gets accessed or modified again, because process Q will immediately page fault and
gets stuck when trying to acquire lock (ft_spt_lock) held by p.

>> B6: (2 marks)
>> A page fault in process P can cause another process Q's frame
>> to be evicted. How do you avoid a race between P evicting Q's
>> frame and Q faulting the page back in?

As mentioned above, when process Q page fault, it gets stuck when acquiring ft_spt_lock, which is held by process P until the eviction process finishes, therefore there is no chance for the eviction process and load process happening at the same time.

>> B7: (2 marks)
>> Explain how you handle access to paged-out user pages that
>> occur during system calls.

During the system calls, the address will be examined by function examine_user_addr(const void*), if the address turns out to be in one of the

user page, the function will look into the page_src of that page and load it back into memory if necessary, and pin down the frame so that it wouldn’t be evicted during the system call, then the rest of system call continues to proceed.

---- RATIONALE ----

>> B8: (2 marks)
>> There is an obvious trade-off between parallelism and the complexity
>> of your synchronisation methods. Explain where your design falls
>> along this continuum and why you chose to design it this way.

We only used one lock for the whole system of all suplementary page tables and frame table, so our design falls into simplicity side. Because we would do swap in eviction, we may change the content of suplementary page table entry of another different process. If we use "smaller" locks, it is possible for some dead lock or race condition to occur.

MEMORY MAPPED FILES

==========

---- DATA STRUCTURES ----

>> C1: (1 mark)
>> Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration that relates to your file mapping table.
>> Identify the purpose of each in roughly 25 words.

added enumerations to 'enum page_src' in 'supt_pt.h':

```
enum page_src
{
        ...
        SRC_FILE,
        SRC_M_FILE,
        ...
};
```

SRC_FILE: To identify that the file is not yet loaded due to lazy loading.

SRC_M_FILE: To identify that the file has been mapped onto memory.

added typedef 'mapped_page' to represent the individual pages in a memory mapping:

```
typedef struct _mapped_page
{
 void* kpage;                    //for SRC_M_FILE
 struct file* f;
 int size;
 size_t ofs;
 size_t page_read_bytes;
}mapped_page;
```

mapped_page: It records the information of that page corresponding to its memory mapping, such as the kernel page, if it has been loaded, the pointer to the file it maps, size of that file and this page's offset to the start of the file (in multiples of page's size), along with the bytes this page should read from the file.

added member 'mapped_page m_page' to union sup_page_info:

```
union sup_page_info
{
  ...
  mapped_page m_page;              // for SRC_FILE/SRC_M_FILE.
  ...
};
```

mapped_page m_page: When 'enum page_src src' in the 'sup_pt_elem' is found to be 'SRC_FILE'/'SRC_M_FILE', the relevant information about its memory mapping can be found in this member of the union.

Added typedef 'mapping_elem' to represent a single memory mapping that a process has:

```
typedef struct _mapping_elem
{
 struct file* f;
 int size;
 int mapping_idx;
 const void* upage;
 struct list_elem elem;
}mapping_elem;
```

mapping_elem: It records the information of the file that it maps, including the pointer and size of that file. It also includes the mapping index returned by the 'mmap' function, the first page of its mapping and also the 'struct list_elem' for it to be stored in 'struct list'.

Added typedef 'mappings' to represent all the mappings that a process has:

```
typedef struct _mappings
{
 struct bitmap* free_map;
 mapping_elem mappings[MAPPINGS_NUM];
}mappings;
```

mappings: 'free_map' records the vacancy of the mapping indices, and
'mappings' stores all mapping_elem mapped by a process, which together can store up to MAPPINGS_NUM of mappings.

PS. To efficiently use the memory allocated to this struct ‘mappings', which takes up the size of one exact page, and also maximize the total number of mappings that a process can own, there would be MAPPINGS_NUM mappings at most, where MAPPINGS_NUM is the number of 'mapping_elem' that can be stored in the remaining page with bitmap storing MAPPINGS_NUM elements deducted from that page.

added member to 'struct thread':

```
struct thread
{
  #ifdef VM
    ...
    mappings* map_info;
  #endif
}
```

mappings* map_info: It records all mappings this process owns.

---- ALGORITHMS ----

>> C2: (3 marks)
>> Explain how you determine whether a new file mapping overlaps with
>> any existing segment. How might this interact with stack growth?

We will traverse all the page that is required to map the file into memory, and check if the page is in suplementary page table(already being occupied by existing memory) and if the page is above LOWEST_STACK_TOP(may overlap with stack or be a kernel addresss). If it is, return error since that page is already being occupied.

---- RATIONALE ----

>> C4: (1 mark)
>> Mappings created with "mmap" have similar semantics to those of
>> data demand-paged from executables. How does your codebase take
>> advantage of this?

First thing we did was to add a union in the struct of supplementary page table's element, so that when we load them lazily, we can distinguish the mappings from the executables by checking the enumeration of the source we are loading from, which is also stored in the element's struct('enum page_src src'). This design, therefore, helps us to cut down duplicate code when installing the page at lazy loading.