Design Documentation

Question 1: Briey describe the data structure(s) that your kernel uses to manage the allocation of physical memory. What information is recorded in this data structure? When your VM system is initialized, how is the information in this data structure initialized?
Answer: We used a structure called frame like:

```
Struct frame {
paddr_t paddr;
int index;
int status;
int group;
};

struct array *frame_table;
struct array *free_frames;
```

The physical address paddr records the physical address of each frame(physical page), int index records the index of each frame in the array frame_table, int status records whether the frame is free or has been used, int group records the group for thr frame using by allocating and freeing contiguous pages. The frame_table is an array of frames to store all the frame information.The free_frames is an array of integers to store all the free frames' index.

When VM system initialized, set the paddr for each frame by updating the first available physical address in the physical memory with PAGE_SIZE, set each frame to free, set group index to 0. And add every frame into the frame_table array, add every frame's index into free_frame array.

Question 2: When a single physical frame needs to be allocated, how does your kernel use the above data structure to choose a frame to allocate? When a physical frame is freed, how does your kernel update the above data structure to support this?
Answer: When a single physical frame needs to be allocated, we just find the first element in the array free_frames which is the first free frames in the physical memory. And use this index to get the frame from the array frame_table, then remove the index from free_frames. And update the frame's status and group, return the paddr.

When a physical frame is freed, we can just set it's status to free and add it's index to the array free_frames.

Question 3: Does your physical-memory system have to handle requests to allocate/free multiple (physically) contiguous frames? Under what circumstances? How does your physical-memory manager support this?
Answer: Our physical-memory system have to handle requests to allocate/free multiple (physically) contiguous frames. When the file size we need to allocate is larger than the PAGE_SIZE.

Since our structure has member group, we can just scan the whole frame_table to find contiguous free frames and update their status, remove their index from the free_frames. Then set their group members to the same group.

Question 4: Are there any synchronization issues that arise when the above data structures are used? Why or why not?
Answer: Yes, there are some synchronization issues. Since the array of frame_table and free_frames are accessed globally, there may be some interrupts when different threads accessing the tables at the same time.

Question 5: Briey describe the data structure(s) that your kernel uses to describe the virtual addressspace of each process. What information is recorded about each address space?
Answer: We used the data structure like:

```
vaddr_t as_vbase1;
paddr_t as_pbase1;
size_t as_npages1;
off_t as_offset1;
int as_permission1;

vaddr_t as_vbase2;
paddr_t as_pbase2;
size_t as_npages2;
off_t as_offset2;
int as_permission2;
paddr_t as_stackbase;

struct array *page_table;
char *elf_file;
```

It records the virtual address, physical address, size, offset and rwx permissions of the code, data segments, the physical address of the stack segment. The page_table is an array of pages to record all the virtual address information. The elf_file records the elf file load to the addressspace.

```
struct page {
vaddr_t vaddr;
paddr_t paddr;
int status;
off_t offset;
int permission;
};
```

This is the structure of the virtual page, it records the page's virtual address, physical address, free or not, offset and rwx permission.

Question 6: When your kernel handles a TLB miss, how does it determine whether the required page is already loaded into memory?
Answer: Since we have the page_table to record all the information about virtual pages. When a TLB miss occurres, the vm_fault will find which page the bad_address belongs to via the index of current addressspace's page_table. Then if the page's status is free, it means the page has not been loaded into memory yet.

Question 7: If, on a TLB miss, your kernel determines that the required page is not in memory, how does it determine where to find the page?
Answer: Since we have the member elf_file in the addressspace structure, then we can just load the page from that elf_file.

Question 8: How does your kernel ensure that read-only pages are not modi_ed?
Answer: In the vm_fault handle we set a integer segment which is initialized when we find the page index of the bad_address, at that time we can recognize the bad_address in code, data or stack and set the segment respectively. If it's in code segment which is read only, then we won't load the page into memory even we can find the VPN in our page_table.

Question 9: Briey describe the data structure(s) that your kernel uses to manage the swap _le. What information is recorded and why? Are there any synchronization issues that need to be handled? If so what are they and how were they handled?
Answer: Not implemented. Maybe we can use a swapfile table to record the physical address and offset of all the pages. And use two method swapin and swapout to load the files.

Question 10: What page replacement algorithm did you implement? Why did you choose this algorithm?
Is this a good choice, why or why not? What were some of the issues you encountered when trying to design and implement this algorithm?
Answer: Not implemented. LFU is our choice. Replacing least frequently used pages would possibly make the most common used programs run faster. It is a good choice because it maintains those common used programs' runtime. Moreover, LRU is needs kernel to concern time and hard to implement. And optimal page replacement is impossible because it requires knowledge of future. However, LFU is really hard for new program to join, therefore, a periodic reducing count of all in-memory page may need.