Group #14

1. We use coremap to efficienly manage physical memory. Coremap structure looks like this:

```
typedef enum _frame_state{
	FREE, USED,		// FREE(Frame currently not used), USED(Frame mapped by page)
} frame_state;

typedef enum _frame_owner{
	NO, KERNEL, USER,	// NO(No Owner. Free frame), KERNEL(Kernel frame), USER(User frame)
} frame_owner;

struct coremap {
	paddr_t pa;		// physical address
	size_t size;		// size of frame
	frame_state state;	// status of frame
	  frame_owner owner;
	int page_num;		// number of frames associated with one another
};

struct coremap *coremaps;
```

Each frame is represented as 'struct coremap' and entire physical memory is represented as a list of coremaps.
When VM system is initialized, coremap is initialized in vm_bootstrap() in vm.c. In vm_bootstrap(), all coremap
in 'coremaps' are assigned with appropriate physical address, FREE states, and size=page_num=0

2. We for-loop 'coremaps' to find 'FREE'-state coremap. When we find empty slot, we set state=USED and set the owner of that frame.
When we want a certain frame to be freed, we calculate the index of coremap and set state=FREE and bzero the data space.

3. Yes. In case when we wish to allocate space that is larger than PAGE_SIZE, we just look for contiguous coremaps in
'coremaps' and if we find right 'chunk', we allocate/free the chunk.
When there are more than two pages allocated, we save the information of how many pages that coremap have.
It is important because when we free the address, we must know how many pages we allocated.

4. Using a synchronization is essential when dealing with a shared resource like coremap. Coremap resides at physical
memory and only one copy of coremap exists. Hence, to avoid conflicts by multiple processors who try to access coremap
at the same time, implementing synchronization is must-do work.

5. struct addrspace {
/* Put stuff here for your VM system */
vaddr_t as_vbase1;
size_t as_npages1;
int as_flag1;
vaddr_t as_vbase2;
size_t as_npages2;
int as_flag2;
/*for load segment */
struct vnode *vn;
off_t offset1;
off_t offset2;
size_t filesz1;
size_t filesz2;
int is_exec1;
int is_exec2;
/* declare page tables */
struct pte *pt1, *pt2, *pt3; // 3 page tables; one for each segment
};

The address space structure contains the virtual base for the code and data segments, which are the vbase1 and vbase2. For each segment, we keep a flag to check for the readable, writable and executable bit. The npages1 and npages2 are the number of pages we have for segment 1 and segment 2 respectively.
The load segment part is the information we save from load_elf function so that when we do on-demand paging later in vm_fault, we can use that information to load segment for the page that is called.
Lastly, we keep 3 page tables, one for each segment(code, data, stack). The page table is an array of page table entries.

6. When the program enters vm_fault on a TLB miss, we check the page table to see if the page requested is valid or not.
If the page was invalid, we have a Boolean flag called “load” which we make true. If “load” is true, we load the page
Into memory, if “load” is false, it means that the page was valid (already loaded into memory) so we do not load the
Page into memory.

7. The load segment information we saved onto the address space is used to find where the page is. We load the segment
from the vnode we saved on the address space, calculate the offset for the page and load PAGE_SIZE amount of file into
memory.

8. In order to make sure the read-only pages are not modified, we set the dirty bit of the page table entry to be zero.
The read-only status of the page is the found by the flag we have in the address space segment. The TLB’s dirty bit is also

Set to zero accordingly so that an exception will be raised if we get a faulttype VM_FAULT_WRITE when the page requested
Is read-only.

9. No structure is used to manage swap file. We used an array of boolean that checks if sectors in swap file
are occupied or not by page table entries. Since each page table entries hold unique index of swap file, no
other info is required. We also applied synchronization since swap file is a shared resource. It's located
in physical memory and only one swap file exists. Hence, we need sychronization when reading from swap file or
write to swap file to prevent any possible errors over the shared resource.

10. We use FIFO algorithm throughout this assignment. The reason we use FIFO is that it's nodoubtfully the easiest
algorithm to implement compared to optimal or LRU or other page replacement algorithms we learned. Since the
assignment specification did not require us to focus on efficiency on the algorithm, we just picked the simplest
algorithm.