Project 4A:

This part of the project is to build a memory allocator for the kernel in xv6. We need to create a new file called kmalloc.c which is very similar to umalloc.c.

First, I looked through the umalloc.c file and tried to understand the functions there. Based on the instruction, we need to modify the morecore() and malloc() calls from umalloc.c file and reuse them in the new kmalloc.file to ask the operating system for the memory.

With the morecore() function , instead of using sbrk() in umalloc.c file, we use kalloc() call in kmalloc.c file. Because sbrk() call allocates a minimum of 4096 units of size, sizeof(Header). To save memory, we can not use sbrk() call. With kalloc() call, It can not return more than 4096 bytes at a time. The following is the modified morecore() function:

```
static Header*
morecore()
{
  char *p;
  Header *hp;
  p = kalloc(); //receive a free page to refill the free chunks
  if(!p)
    return 0;
  hp = (Header*)p;
  hp->s.size = 4096;
  kmfree((void*)(hp + 1));
  return freep;
}
```

With the kmalloc() function, we assume the call to kmalloc() is limited to 4096 bytes(a single page) and add a panic if the request is greater than 4096 bytes. The following is the modified kmalloc:

```
void*
kmalloc(uint nbytes)
{
  //panic to check if the request is greater than a page in length
  if (nbytes > 4096)
    panic("kmalloc does not allocate more than 4096 bytes.");
 //the following code is copied from malloc() in umalloc.c
}
```

The code for kmfree() does not change. The kmfree() assumes the free list is in increasing order of addresses within the kernel space. If the chunk to be freed is close to a chunk in the list, it combines with the close chunk. There is a chance for the chunk to combine with two close chunks on the list. If the chunk can not be combined, then it's placed at the appropriate location within the list.

To define the two new system calls for kmfree and kmalloc, we will need to make changes to syscall.h, syscall.c, sysproc.c, usys.S and user.h files.

In sysproc.c, adding the following codes to define these two system call functions:

```
int
sys_kmfree(void)
{
  void *addr;
  argint(0, (int*)&addr);
  kmfree(addr);
  return 0;
}

int
sys_kmalloc(void)
{
  int bytes;
  argint(0, &bytes);
  return (int)kmalloc(bytes);
}
```

Project 4B:

In this part of the project, we will create a data structure used to keep track of the memory mappings allocated by mmap. The data structure we will use is a simple linked list that maintains an increasing order of addresses for the mapped pages. I added the data structure in the proc.h:

```
//data structure to keep track of mmap regions
struct mmap_region {
    void *addr;
    int length;
    int region_type;      //anonymous vs field-backed
    int offset;       // the offset into the file for file-backed memory regions
    int fd;
    struct mmap_region *next;
};
```

With mmap.c, the mmap() function creates a new mapping for the calling process's address space. It returns the starting address of the newly mapped memory region or (void*)-1 on failure. First, we check if the allocated memory for the current process's size is zero or not, if it's zero, we return (void*) -1. Then we switch TSS and the hardware page table to the current process. Next, we create a structure object new_mmap_region and allocate memory for list nodes by using kmalloc() from the kmalloc.c file. Then assigning list data and metadata to the new_mmap_region. If the current process's mmap's size is equal to zero, then we save the new_mmap_region as the head. If it's not, we add regions to the existing mapped regions list and we increase the mmap_sz. Then return the address of the newly mapped memory region.

```
void *mmap(void *addr, int length, int prot, int flags, int fd, int offset)
{
    //get pointer to current process
    struct proc *curproc = myproc();
    int address = curproc->sz;
```

```
    if((curproc->sz = allocuvm(curproc->pgdir, address, address + length)) == 0)
        return (void*)-1;
    switchuvm(curproc);
    cprintf("After switchuvm\n");
    //List nodes are allocated using kmalloc (from kmalloc.c)
    struct mmap_region *new_mmap_region = kmalloc(sizeof(struct mmap_region));
    cprintf("After kmalloc\n");
    //assign list data and metadata to the new region
    new_mmap_region->addr = (void*)PGROUNDDOWN((uint)address);
    new_mmap_region->length = length;
    new_mmap_region->fd = fd;
    new_mmap_region->offset = offset;
    new_mmap_region->next = 0;
    cprintf("After assigning mmap \n");
    // first region, save as head
    if(curproc->mmap_sz == 0)
    {
        curproc->mmap_hd = new_mmap_region;
    }
    else  //add region to existing mapped_regions list
    {
        struct mmap_region *cursor = curproc->mmap_hd;
        while (cursor->next)
        {
            cursor = cursor->next;
        }
        cursor->next = new_mmap_region;
    }
    curproc->mmap_sz += 1;
    return new_mmap_region->addr;
}
```

Initializing the mmap size and list in the allocproc() in proc.c file.

```
p->mmap_sz = 0;
p->mmap_hd = 0;
```

With munmap() function, first, we check the current process's mmap size. If it's zero, we return -1. Because there is nothing that has been allocated. Then we check the head of the linked list. If the head's address and length are equal to the given address and length, then we found one valid mapping. If the head's address and length are not equal to the given address and length, then we move to the next node and compare the node's address and length with the given address and length. If they are equal, then find = 1. If we found mapped regions, first, then we use deallocuvm() to unmap the memory region. If deallocuvm is equal to zero, then we deallocate the data structure and free memory regions before returning -1. Finally, we could use kmfree() to free data structure elements used to track the freed region and return zero as a success.

```
int munmap(void *addr, uint length)
{
    struct proc *curproc = myproc();
    int address = curproc->sz;
    //if nothing has been allocated, there is nothing to munmap
    if (curproc->mmap_sz == 0)
    {
        return -1;
    }
    struct mmap_region *head = curproc->mmap_hd;
    struct mmap_region *itr = head;
    struct mmap_region *tmp;
    int find = 0;
    if(head->addr == addr && head->length == length)
    {
        tmp = head;
        curproc->mmap_hd = head->next;
        find = 1;
    }
    else
    {
        while(itr->next)
        {
            if(itr->next->addr == addr && itr->next->length == length)
            {
                tmp = itr->next;
                itr->next = itr->next->next;
                find = 1;
                break;
            }
            itr = itr->next;
        }
    }
    if(find)
    {
        //deallocate the memory from the current process
        if((curproc->sz = deallocuvm(curproc->pgdir, address, address - length)) == 0)
        {
            free_mmap_regions(curproc->mmap_hd);
            return -1;
        }
        switchuvm(curproc);
        kmfree(tmp);
        curproc->mmap_sz -= 1;
```

```
        return 0;  //return success
    }
    free_mmap_regions(curproc->mmap_hd);
    return -1;
}
```

I also create a helper function to recursively free map regions :

```
void free_mmap_regions(struct mmap_region *mmap_region)
{
    if(mmap_region && mmap_region->next)
        free_mmap_regions(mmap_region->next);
    if (mmap_region)
        kmfree(mmap_region);
}
```

Then, I added the following line in the freevm() function in vm.c file:

```
 free_mmap_regions(myproc()->mmap_hd);
```

As usual, when we create new system calls, we need to modify five files: syscall.h, syscall.c, sysproc.c, usys.S, and user.h files.

Reference:
https://medium.com/@viduniwickramarachchi/add-a-new-system-call-in-xv6-5486c2437573