Project 5A:

In this part of the project, we need to change the current behavior of mmap() from immediate allocation (kalloc()) and mapping (mappages()) of physical memory pages to lazy page allocation, where pages are only allocated when a page fault occurs during access.

In mmap.c(), we need to delete the switchuvm(curproc) for not loading the process's page table and we do not need to call allocuvm() when mmap is being called. We only allocate the regions when the page fault occurs.

To handle page faults, we need to modify the trap.c file. When a trap is generated in xv6, it is handled in the switch statement inside the trap.c. Therefore, we should add some codes to check if the trap number corresponds to a page fault which is stored in the traps.h file as T_PGFLT.

```
case T_PGFLT: //added page fault check in the trap switch statement
    pagefault_handler(tf);
    break;
```

Next, we need to create a page fault handler to attempt to recover from a page fault. If the trap number is equal to T_PGFLT, then call pagefault_handler() and the mmap linked list will be searched to check if the faulting address has been declared as allocated. If it has been allocated, the page containing the faulting address will be allocated to the process. The following is the pagefault_handler() function:

```
void
pagefault_handler(struct trapframe *tf)
{
  struct proc *curproc = myproc();
  //control Register 2, holds the faulting page address.
  uint fault_addr = PGROUNDDOWN(rcr2());
  //debugging statements from the instruction
  cprintf("============in pagefault_handler============\n");
  cprintf("pid %d %s: trap %d err %d on cpu %d "
  "eip 0x%x addr 0x%x\n",
  curproc->pid, curproc->name, tf->trapno,
  tf->err, cpuid(), tf->eip, fault_addr);
  //if the faulting address belongs to a valid mmap region
  int find = 0;
  mmap_region *head = curproc->mmap_hd;
  while(head)
  {
    if ((uint)(head->start_addr) <= fault_addr && (uint)(head->start_addr +
head->length) > fault_addr)
    {
       if ((head->prot & PROT_WRITE) || !(tf->err & T_ERR_PGFLT_W))
      {
        find = 1;
        break;
      }
```

```
      head = head->next;
    }
  }
  // mapping a single page around the faulting address, used the allocuvm()
  if (find == 1)
  {
    char *memory = kalloc();
    if(memory == 0)
    {
      goto error;
    }
    memset(memory, 0, PGSIZE);
    // if protection bits needed for mappages()
    int permissions;
    if (head->prot == PROT_WRITE)
    {
      permissions = PTE_W|PTE_U; //give write permissions
    }
    else
    {
      permissions = PTE_U; //No write permissions
    }
    if(mappages(curproc->pgdir, (char*)fault_addr, PGSIZE, V2P(memory), permissions) <
0)
    {
      kfree(memory);
      goto error;
    }
    switchuvm(curproc);
//end allocuvm
```

To use mappages, we need to remove the static from the mappages function in vm.c and declare mappages() in defs.h:

```
int             mappages(pde_t*, void*, uint, uint, int);
```

Project 5B:

The second part of the project is to support file-backed mmap regions.

First, we add a function called fileseek() in file.c which allows the user to change the offset field in a file struct. By referencing the filewrite() function in the file.c, we first need to acquire the inode lock before modifying the offset field. The fileseek() function shows below:

```
int
fileseek (struct file* f, uint offset)
{
  begin_op();
  ilock(f->ip); //acquire inode lock
  f->off = offset;
```

```
  iunlock(f->ip);
  end_op();
  return 0;
}
```

mmap() also takes in protection bits and flags to customize the behavior of the memory region. Based on the instruction, we need to add the support for protection bits and few flags for mmap(). We create a new file called mman.h.

```
// Protection bits
#define PROT_WRITE      1
// Flags
#define MAP_ANONYMOUS   0
#define MAP_FILE        1
```

Next, we need to extend our mmap() and munmap() implementation to handle the arguments (int fd, int offset).
In the mmap() function, we first check if the flag is equal to MAP_ANONYMOUS. If fd is not equal to -1, then we free the memory regions and return -1. Then, we check when the flag is equal to MAP_FILE, if fd is greater than -1, then we create a duplicate file descriptor to save in the data structure. If fd is not greater than -1, we free the memory regions before returning -1.
In munmap() function, we need to deallocate the duplicated file descriptor and decrement the open count using fileclose() before freeing the data structure.

```
//check the flags and file descriptor argument
  if (flags == MAP_ANONYMOUS)
  {
    if (fd != -1) //fd must be -1
    {
      kmfree(new_mapp_region);
      return (void*)-1;
    }
  }
  else if (flags == MAP_FILE)
  {
    if (fd > -1)
    {
      if((fd=fdalloc(curpro->ofile[fd])) < 0)
        return (void*)-1;
      filedup(curpro->ofile[fd]);
      new_mapp_region->fd = fd;
    }
    else
    {
       kmfree(new_mapp_region);
      return (void*)-1;
    }
  }
```

Msync() function syncs the changes made to a file in memory back to the external file thus solidifying the changes that are being made through memory manipulation. If the function is not returned successfully, the changes made in a file-backed mmap are not guaranteed to stay in the memory.

```
int msync (void* start_addr, uint length)
{ struct proc *curproc = myproc();
  // If nothing has been allocated, no need to do msync
  if (curproc->nregions == 0)
  {
    return -1;
  }
  mmap_region *itr = curproc->mmap_hd;
  while(itr)
  {
    if(itr->start_addr == start_addr && itr->length == length)
    {  //check the address was allocated
      pte_t* ret = walkpgdir(curproc->pgdir, start_addr, 0);
      if((uint)ret & PTE_D){
        //do nothing
      }
      fileseek(curproc->ofile[itr->fd], itr->offset);
      filewrite(curproc->ofile[itr->fd], start_addr, length);
      return 0;
    }
    itr = itr->next;
  }
  return -1; //No match
}
```

In order to optimize msync(), we can check for the dirty bit of page table entries.

```
#define PTE_D           0x040   // Dirty
```

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

```
int sys_msync(void)
{
  int addr;
  int length;
  if(argint(0, &addr) < 0){
    return -1;
  }
  if(argint(1, &length) < 0){
    return -1;
  }
  return msync((void*)addr, (uint)length);
}
```