Project 3A

First, I followed the instructions to make a program that dereferences a null pointer. I took an example from the stackoverflow:
https://stackoverflow.com/questions/4007268/what-exactly-is-meant-by-de-referencing-a-null-pointer

```
#include "types.h"
#include "stat.h"
#include "user.h"
#define NULL((void *)0)
int main(int argc, char *argv[]){
   int a, b;
   int *pi;
   a = 5;
   pi = &5;
   a = *pi;
   pi = NULL;
   b = *pi;
   printf(1, "Null Pointer value: %p\n", *pi);
   exit();
}
```

I run this program in the xv6, and just like the instruction said there is no exception, and I got the Null Pointer value : 83E58955.

The second of this project is to make two new system calls. I followed the instructions in this article to make the changes in the files for new system calls:
https://medium.com/@viduniwickramarachchi/add-a-new-system-call-in-xv6-5486c2437573

First, I went to the vm.c file to look at how the page table is being set up and how exec() and fork() works. In the instruction, it asks us to change the protection bits of parts of the page table. So I decided to write these two functions int mprotect(void *addr , int len) and int munprotect(void *addr, int len) in the vm.c file.

In mprotect() function, I first check if the addr is page aligned or not and also if the len is less than or equal to zero. Then get the running process and get the physical address of page table entry using virtual address curr and clear writable bit of page table entry:

```
pte = walkpgdir(curproc->pgdir, (void *)curr , 0);
*pte &= ~(PTE_W);
curr += PGSIZE;
```

Next, based on the hint in the instruction, we need to update the page table entry to let hardware know.

Function mumprotect() is very similar with mprotect(), except we need to set the writable back:

```
pte = walkpgdir(curproc->pgdir, (void *)curr , 0);
   *pte |= (PTE_W);
   curr += PGSIZE;
```

Since address 0 is now used for null dereference check, we need to change the for loop(line 328) starts from the PAGESIZE and also change the code line 150 in the MakeFile entry point 0

to 0x1000 and the sz(on line 43) equals to PGSIZE in the exec.c file. Because I wrote two new functions in the vm.c file, I need to add these two lines in the defs.h file:

int mprotect(void, int);*

int munprotect(void, int);*

Next, I followed the steps from the article that posted above to add the two new system calls: First , I went to systcall.h to add:

#define SYS_mprotect 22

#define SYS_munprotect 23

Then, I added the following codes in the syscall.c *: extern int sys_mprotect(void);*

extern int sys_munprotect(void);

[SYS_mprotect] sys_mprotect,

[SYS_munprotect] sys_munprotect,

When the system call with number 22 is called by a user program, the function pointer sys_mprotect(void) which has the index SYS_mprotect or 22 will call the system call function. Next, we will implement the system call function. I defined the two system calls in the sysproc.c file.

For a user program to call the system call, an interface needs to be added. Therefore, I edited the usys.S file with the following code:

SYSCALL(mprotect)

SYSCALL(munprotect)

Last, the user.h file needs to be edited.

int mprotect(void, int);*

int munprotect(void, int);*

Project 3B

I followed the instructions to finish this project. We need to work on three things. The first one is to define a new system called clone() to create a kernel thread. The second one is to build a system call join() to wait for a thread. The third one is to build a thread library.
This article is a good reference of what files you need to work on if you want to add a system call in xv6:
https://medium.com/@viduniwickramarachchi/add-a-new-system-call-in-xv6-5486c2437573

System call clone():

First, I went to the proc.c file to look at the fork() system call. Then I started writing a clone() system call based on fork(). We need to check the page alignment, and the stack should be one page in size. Just like the fork(), we need to allocate the process and copy the process state from the process. However, fork() is to copy the address space and point the page directory to a new page table. In clone(), the child shares the same address space with the parent and has the same page table. When clone() returns in the newly created thread, it runs on the user stack, so I set up the user stack with two arguments and set the top of the stack to the allocated page. Then, I copy user stack values to the np's memory.

System call join():

I also added this function to the proc.c file.
First, we need to scan through the table looking for the child threads that belong to the parent process. If we find one, we will "kill" it by setting it to the UNUNSED state and resetting all of its values. It will then return the PID of the child thread that was killed. If we did not find any child, there's no need to wait.
Since I added two new system calls, I went to syscal.h file and added two line:

```
#define SYS_clone  22
#define SYS_join   23
```

Next, we need to add a pointer to the system call in the syscall.c file.

```
[SYS_clone]   sys_clone,
[SYS_join]    sys_join ,
```

When the system call with number 22 is called by a user program, the function pointer sys_clone which has the index SYS_clone or 22 will call the system call function.
Next, we need to define these system call functions in the sysproc.c file, so we can implement them.
Then we need to add an interface for the user program to call the system call. I added these two lines in the usys.S file:

```
SYSCALL(clone)
SYSCALL(join)
```

Then, I went to the user.h to add these lines of code:

```
int thread_create(void (*)(void*, void*), void *, void *);
int thread_join();
```

I also add these two lines of codes in the defs.h:

```
int             clone(void(*)(void*, void*), void*, void*, void*);
int             join(void**);
```

Thread Library:

I put this part of the code in the ulib.c file.

I reference the chapter 28(locks) and https://en.wikipedia.org/wiki/Fetch-and-add for the ticket lock. First, initialize lock to 0(0 means the lock is available) and then implement an atomic fetch-and-add operation in the lock_acquire(spin until the lock is acquired). Then release the lock and set it to 0 again.

In thread_create(), we acquire lock first, then use malloc() to create a new user stack, next we release the lock. Then we call clone() to create the child thread and return it.

In thread_join(), we call join() system call, then acquire the lock, free the user stack, release the lock, then return.

Since we add the new thread library in the ulib.c, we need to add the function initialization in the user.h as well. Also, we need to declare the type lock in the types.h:

```
typedef struct __lock_t {
    uint ticket;
} lock_t;
```