Programming Assignment 1 Report
Aayush Patel - CS20BTECH11001

The first task of the assignment involved printing the name and return value system calls executed when the xv6 kernel is booted.
Let me explain how a system call is executed(or atleast what I understood from this assignement):

1. A system call is executed in kernel mode so that it has some extra priviledges. The program needs access to some low level functionalities. The program does so by generating a trap.
2. Look at the *usys.S* file. The assembly file describes a macro for all the system calls. Labels are created for each syscall with name *SYS_{SYSCALL_NAME}*. For each of the syscall, the **eax** register is set to the number assigned to that particular syscall. We will talk more about this number later.
3. The **int** instruction in the *usys.S* file generates a software interrupt and sets the interrupt number to the macro T_SYSCALL. Since traps are generated for various other situtations(like say divide by zero), the interrupt number T_SYSCALL will convey to the interrupt handler that the interrupt was generated to execute a syscall.
4. All traps are handled in *trapasm.S* assembly file. At line 20, the given file makes a call to the C **trap(struct trapframe *)** function in the *trap.c* file with parameters as a pointer to the **struct trapframe**.
5. Notice that at line 39 of *trap.c,* the trap number is checked with T_SYSCALL which we discussed in point 3. At line 42, the process' trapframe is assigned to the trapframe sent as parameter and then finally at line 43, the **syscall()** function is called in *syscall.c* file.
6. At line 158 of the *syscall.c* file, we retrive the value which we stored in the **eax** register(we discussed this in point 2) and store it in **num.** This number will help us identify which syscall is executed as every syscall is assigned a unique number. Then some check are made at line 159 to see if the number is valid. Then at line 161, the **eax** value of the current processes trapframe struct is overwritten with the return value from the syscall function defined in the *sysproc.c* file. Note that syscall is an array of function pointers. This completes the execution of the system call.
7. For the given assignment, we need to implement a system call which can accept parameters. You will notice that all the syscall handlers in *sysproc.c* have void as parameters. But in reality, syscalls like open do take parameters. Whenever a user function(say **open**) is passed with parameters(filename of file to open in this case), the parameters are stored in the user stack and whenever a syscall handler wants to access the

parameters, **argint**,**argstr** and **argptr**(defined in *syscall.c*) are used. They can be used to retrive the pointers to the memory location storing the variables in the user stack. This can be advantageous in many ways. If we pass an integer as a parameters, unnnecessary copy will be made due to pass by value. Passing a pointer to parameters saves us from this. Also if we look at the implementations of **argint**,**argstr** and **argptr,** checks are made that make sure that the memory location we are trying to access is in the address size allocatted to that process before returning the value.