Write up for Question 2

Steps followed for compiling the kernel and adding system call:

1. Download kernel source code using wget command
2. Extract the tar.gz file
3. Goto to directory kernel/ in the extracted folder
4. Edit file sys.c, ex : sudo nano sys.c
5. Add you syscall code in the file sys.c
6. Go to directory arch/x86/entry/syscalls/ in extracted folder
7. Add your system call to file syscall_64.tbl
   Ex: 440 common sample sys_sample
8. Edit config file so that only required things are compiled
9. Compile the kernel with sudo make -j($number of processors to use)
10. Install modules using : sudo make modules_install install
11. Reboot system

Error handling done in system call :

1. Handling the case when creating task_struct fails

```
if (task == NULL) {
      printk(KERN_ALERT "Cannot find a process with pid %d\n",
pid);
      return -ESRCH;
 }
```

   where task is the name of task_struct pointer.

2. Handling case where strncpy_from_user() fails

```
if (val < 0 || val == n)
      return -EFAULT;
```

where val is the value returned by strncpy_from_user()

3. error handling unable to open file i.e. filp_open fails

```
if (IS_ERR(ffile) || ffile == NULL) {
        printk(KERN_ALERT "ERROR: filp_open %ld\n",
PTR_ERR(ffile));
        return PTR_ERR(ffile);
 }
```

   IS_ERR is used to check if there is any error and PTR_ERR() is used to identify the error.

4. To be able to debug easily the syscall outputs the return values of every *kernel_write()* to the kernel console which can be viewed using the *dmesg* command.

Error handling doing in test.c:

1. Check if the arguments contain at least two inputs.

```
if (argc <= 2) {
    printf("Incorrect Arguments\n");
    return -1;
}
```

2. Identify is that the system call returns any error using perror.

```
perror("Console Output");
```

   Prints “success” if no error is encountered

Short Description of how system calls is written and functions used (see more in comments) :

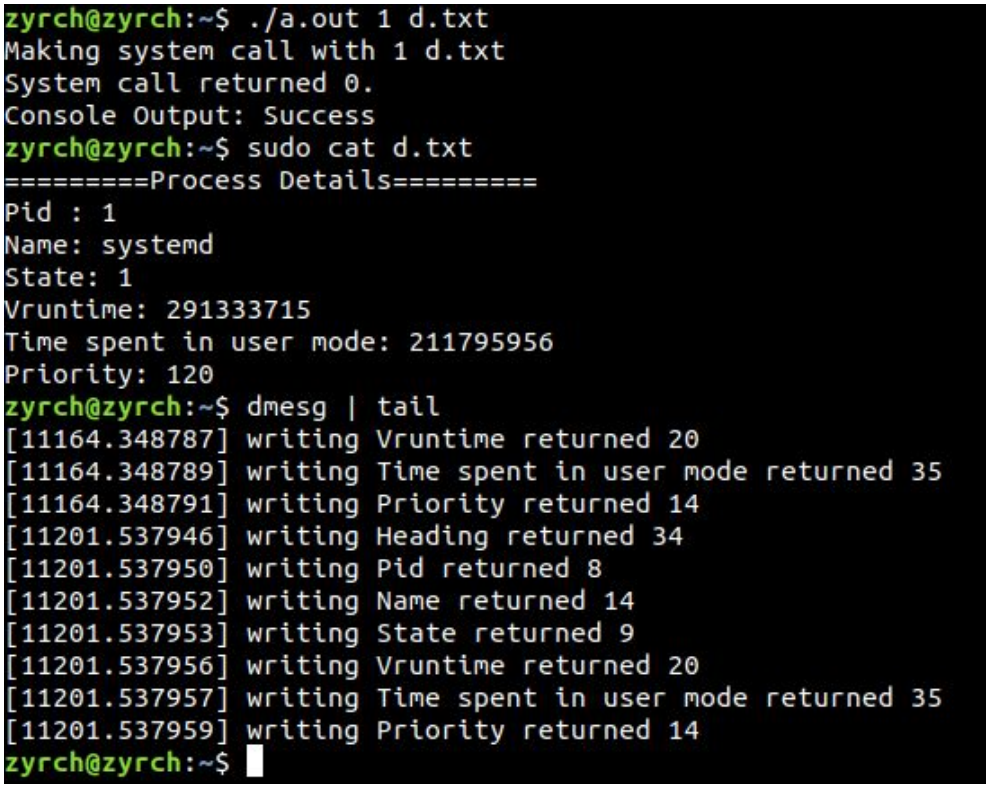
In the system call file all the information about the process is obtained using its pid in a *task_struct* using *pid_task()* and *find_get_pid()*. Then the fileName is first copied to kernel memory space and then its opened using *filp_open()*.

Now the next step is to write the file from kernel, to achieve this *kernel_write()* is uses a char * buffer. To write desired text into a char * buffer sprintf is used, which works as standard printf except it outputs the result to a buffer provided as input instead of stdout console.

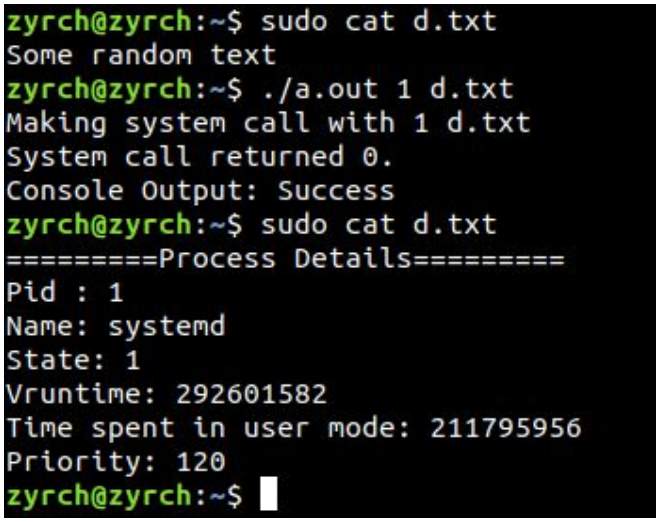
Example Outputs:

Input format : ./a.out $pid $filename

1. Printing info about process with pid 1 to d.txt



2. If a file already exists then it's truncated.



3. If invalid pid is entered error is outputted accordingly