CSE231 – Operating Systems
Assignment 2 – Q2
Writeup

Name: Ansh Arora
Roll no: 2019022

Task : To write a syscall in C for the linux kernel sh_task_info() which takes a PID as argument and prints out the given PID's process's details on to the terminal and also stores the corresponding fields onto a file with a filename that has been given by the user.

Description of code and implementation -

The system call has been made using the SYSCALL_DEFINEn() function, which is capable of taking in n parameters. Since I take in 3 parameters from my test C file into the sys call, I have use SYSCALL_DEFINE3(). The parameters that have been used are – pid (int, passed by user), file_path (char*, given by user), path_length (int, calculated based on the filename given by the user after adding a .txt to its end).
All the variables used in the file are declared at the start of the function code (in accordance with C90's norms).
Here I first copy the filename passed by the user as a pointer into a local string char array to avoid any pointer errors in the program. This is done using the copy_from_user function, that is used to copy data from user space into the kernel space.

After the copying of this data, to find the task_struct of the given PID's process I use two functions in conjunction – find_get_pid() and pid_task(). The find_get_pid() function takes in the pid integer and returns the corresponding process's pid_struct. We then use the PIDTYPE_PID field of this pid_struct in the pid_task() function to get the task_struct of the taken function and store it inside the task_struct variable task.

After this, I use the task_struct variable to find these values of the Process –
1) Process Name – task->comm
2) Process ID – task->pid
3) Process State – task->state
4) Process Priority – task->prio
5) Process Parent Name – task->parent->comm
6) Process Parent ID – task->parent->pid

These values are then displayed onto the kernel's messages (can be viewed either in Ctrl+Alt+F2 mode or by typing dmesg). They are displayed by using printk in conjunction with KERN_ALERT, so that it can be displayed in the kernel's messages.

After this, to add this data into a file, I first open the file using the filp_open() command, where I pass the filename that has been given by the user. The filp_open() command has been used using the O_CREAT flag so that if the given filename doesn't already exist, it will be created and started in the write mode thanks to the O_WRONLY flag.
The data to be stored into the file is passed into a single character string array data[2000] using the scnprintf function in C. All data is combined in the exact same format as it is to be displayed onto the screen and in the textfile.

The data is then passed into the textfile using the kernel_write() command, which takes the file path, data array, data array length and the opened file's pointer as arguments.
After successfully writing the data into the passed name's text file, the file is closed using the filp_close() command, and hence the syscall is ended.

The system call is added to the terminal by first adding it to it to a folder sh_task_info inside the main folder. A Makfile is made withing the folder assigning the object file of the syscall to obj-y. In the main folder's Makefile, the sh_task_info folder is included in the core-y list so that it is considered while building the kernel. At last the syscall is added to syscalls table at #440 as of common type, which is also used to call the syscall in the test.c file.

Code of system call –

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/printk.h>
#include <linux/module.h>
#include <asm/uaccess.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/proc_fs.h>
#include <linux/pid.h>
#include <linux/pid_namespace.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/uaccess.h>


SYSCALL_DEFINE3(sh_task_info,int,pid,char *,file_path,int,path_length){
        //int p_id;
        struct pid *pid_struct = NULL;
        struct task_struct *task;
        int proc_id;
        long int proc_state;
        int proc_prio;
        int proc_parent_id;
        struct file* data_file = NULL;
        char data[1000];
        int datalen;
        char filepath[200];

        if(copy_from_user(filepath,file_path,path_length) != 0){
                return -EFAULT;
        }

        pid_struct = find_get_pid(pid);

        if(pid_struct==NULL){
                return -ESRCH;
        }

        task = pid_task(pid_struct,PIDTYPE_PID);
```

```
        //char proc_name[100] = task->comm;
        proc_id = (int) task->pid;
        proc_state = (long int) task->state;
        proc_prio = (int) task->prio;
        //char proc_parent_name[100] = task->parent->comm;
        proc_parent_id = (int) task->parent->pid;

        printk(KERN_ALERT "Process Name: %s \n",task->comm);
        printk(KERN_ALERT "Process ID: %d \n",proc_id);
        printk(KERN_ALERT "Process State: %ld \n",proc_state);
        printk(KERN_ALERT "Process Priority: %d \n",proc_prio);
        printk(KERN_ALERT "Process Parent Name: %s \n",task->parent->comm);
        printk(KERN_ALERT "Process Parent ID: %d \n",proc_parent_id);


        data_file = filp_open(filepath, O_WRONLY | O_CREAT, 0);

        if(data_file==NULL){
                return -ENOENT;
        }

        scnprintf(data,sizeof(data),"Process Name: %s \nProcess ID: %d
\nProcess State: %ld \nProcess Priority: %d \nProcess Parent Name: %s \nProcess
Parent ID: %d \n",task->comm,proc_id,proc_state,proc_prio,task->parent-
>comm,proc_parent_id);
        datalen = (int) strlen(data);

        kernel_write(data_file,data,datalen,&data_file->f_pos);

        filp_close(data_file,NULL);

    return 0;
}
```

Inputs that user should give –

1. The process ID or pid, when asked for.
2. The textfile's name in which the data will be saved. By default, the textfile is created in the same folder in which the test.c file is held.

After the program ends running, if successful(error less) it displays that the program has returned a code 0. You can now go and view the data printed onto the kernel messages using dmesg, or can view the data by typing cat <filename given>.txt.

Expected Output –

```
ansharora@ubuntu:~/Desktop$ ./a.out
Enter file name: procdata
Enter pid: 11663
Sys call returns 0
```

```
ansharora@ubuntu:~/Desktop$ sudo cat procdata.txt
Process Name: firefox
Process ID: 11663
Process State: 1
Process Priority: 120
Process Parent Name: systemd
Process Parent ID: 1
```

```
[16516.355319] Process Name: firefox
[16516.355323] Process ID: 11663
[16516.355324] Process State: 1
[16516.355324] Process Priority: 120
[16516.355325] Process Parent Name: systemd
[16516.355326] Process Parent ID: 1
```

The details are all shown as was told before. Since this was a successful run, the syscall returned a 0 value and the details were printed out to the kernel messages and to the defined text file.

Error Values and How to Interpret Them –

All errors have been created using the errno standard and errno.h header file. They are returned in the syscall in the egs- return -EIO method.

1. ESRCH – (Errno = 3) No such process found. If a given pid is not able to find a process, ESRCH is

returned. This is handled in the test.c file as an errno generated, and the error number is printed out along with error message using perror() and stderror().

2. ENOENT – (Errno = 2) No such file or directory found. This is printed in case the file asked for is unavailable for some internal reasons. I return the ENOENT value when the filp_open() command is unscuccessful.

3. EFAULT – (Errno = 14) Bad address. This is returned if the filename pointer being used for copy_from_user() function is not working. In this case the copy_from_user() command fails and the syscall is ended with an error and EFAULT value is returned to the test.c file.

Example of Error generated during run –