Operating Systems

Assignment-2

Rohan Jain
2019095

Question 1

Yes, there is a difference between the value of the global variable printed in the two cases. In case 1, `fork()` duplicates the process completely using `COW` principles. However, `pthread_create()` only copies the `code` and `stack` section of the process. This means that both child and parent thread point to the same global variable. In `fork()`, the parent and child processes have their own copies of the global variable.

In the case of `fork()`, if the parent process has counted up to `35` and then context switch happens, the child doesn't start from `35`, it starts from `10`. Suppose the child now counts down to `-1` and context switch happens, the parent will start from its value, i.e. `35` and not `-1`. This is because both processes have their own memory allocated for that global variable.

In the case of `pthread_create()`, if the parent process has counted up to `35` and then context switch happens, the child starts from `35` and not `10`. Suppose the child now counts down to `-1` and context switch happens, the parent now starts from `-1` and not `35`. This is because both processes have only one shared memory for that global variable in the common `data` section.

`fork()` sample:

```
Parent Process: 40
Parent Process: 41
Parent Process: 42
Parent Process: 43
Parent Process: 44
Parent Process: 45
Parent Process: 46
Parent Process: 47
Parent Process: 48
Child process running! PID is: 40364
Parent Process: 49
Parent Process: 50
Child Process: 9
Parent Process: 51
Child Process: 8
Parent Process: 52
Child Process: 7
Parent Process: 53
Child Process: 6
Parent Process: 54
Child Process: 5
Parent Process: 55
Child Process: 4
```

`pthread_create()` sample:

```
Parent Thread running!
Parent Thread: 11
Parent Thread: 12
Parent Thread: 13
Parent Thread: 14
Parent Thread: 15
Parent Thread: 16
Parent Thread: 17
Parent Thread: 18
Parent Thread: 19
Parent Thread: 20
Child Thread created!
Child Thread: 20
Child Thread: 19
Child Thread: 18
Child Thread: 17
Child Thread: 16
Child Thread: 15
Child Thread: 14
Child Thread: 13
Parent Thread: 21
Parent Thread: 13
Parent Thread: 14
Parent Thread: 15
Parent Thread: 16
Parent Thread: 17
Parent Thread: 18
Parent Thread: 19
Parent Thread: 20
Parent Thread: 21
Parent Thread: 22
Child Thread: 12
Child Thread: 22
Child Thread: 21
Child Thread: 20
```

In both cases, the parent thread/process eventually reaches `100` and child thread/process reaches `-90`. However, in case of thread there is a possibility that the program never ends. If the parent increments the number 10 times, context switch happens, and then child process decrements number 10 times and again context switch happens. In this case, the process may never end. The chances of this happening are very less though and the process finishes almost every time with varying time taken.

Question 2

Description

The system call prints PID, State, Real time priority, Scheduling policy, Number of CPUs allowed, Command name and the number of context switches done of the process corresponding to `pid` on kernel log and in a file given by the parameter `filename` in the current working directory.

Function definition

```
long sys_sh_task_info(pid_t pid, char* filename);
```

Returns 0 on success and negative number on error. It sets `errno` on return.

To check the output of the syscall on kernel log: `dmesg | tail`

Explanation of Code:

```
SYSCALL_DEFINE2(sh_task_info, pid_t, pid, char *, file_name)
```

This is the main system call function that is defined using the `SYSCALL_DEFINEn` macro. To find the `task_struct` corresponding to the `pid` passed in parameter I have used functions `find_vpid()` and `pid_task()`.

```
task = pid_task(find_vpid(pid), PIDTYPE_PID);
```

If the `task_struct` corresponding to the `pid` is not found, the `errno` is set to `ESRCH` = No such process and the returned value is `-ESRCH`.

Next, the `set_output()` function is called.

```
int set_output(char *output, struct task_struct *task)
```

The function `set_output` populates the string output with the fields given in description. It takes the fields from the `task_struct` pointed by `task`. It makes use of functions like `scnprintf()` and `strncat()` which are defined in the kernel API.

To access the string passed in filename, I have to copy the string from user space to kernel space. To do this, I have used the function `copy_from_user()`

```
copy_from_user(filepath, file_name, 256);
```

If the `filename` given is greater than `256` bytes, the `errno` is set to `ENAMETOOLONG` = File name too long and the returned value is `-ENAMETOOLONG`.

If the `filename` given is invalid, then errno is set to `ENOENT` = No such file or directory and the returned value is `-ENOENT`.

I open/create the file using `filp_open()`. I give the permissions `0666` and set the flags to write and create.

```
fileptr = filp_open(filepath, O_WRONLY | O_CREAT, 0666);
```

After getting the `fileptr (struct file*)` , I write the previously generated `output` string to the file pointed by `fileptr` using `kernel_write()`.

```
returnVal = kernel_write(fileptr, output, len, &pos);
```

If `kernel_write()` does not write the whole message, then `errno` is set to `EIO` = I/O Error and the returned value is `-EIO`.

I close the file using `filp_close()` and return 0 as everything worked if we have reached this command.

```
filp_close(fileptr, NULL);
```

Expected Input Output

The user has to call the system call and give 2 inputs `pid_t pid` and `char *filename`. `pid` refers to the pid of the process whose details have to be printed and `filename` refers to the name of the file in which the details are stored.

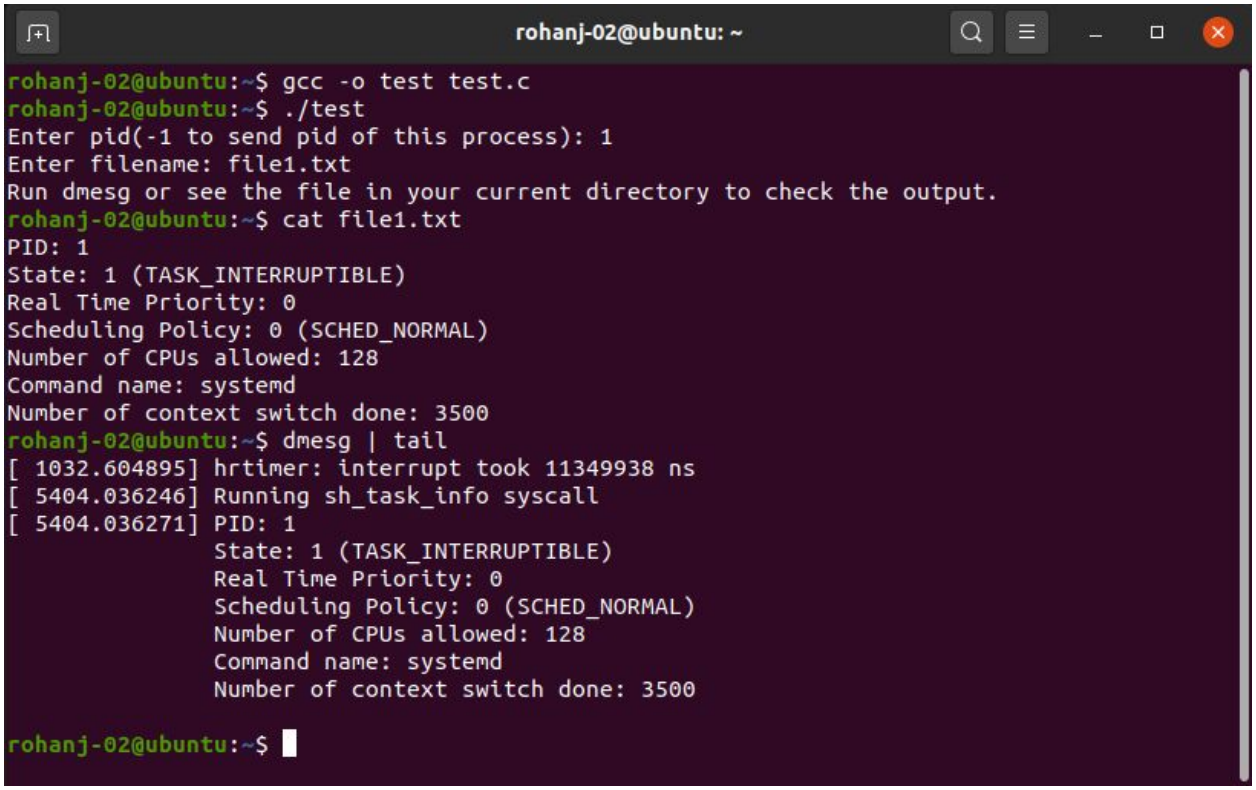
The output of the system call is all the details of the values mentioned in description. The string representations of the numeric values like scheduling policy are also mentioned in the printed values.

All errors returned by code:

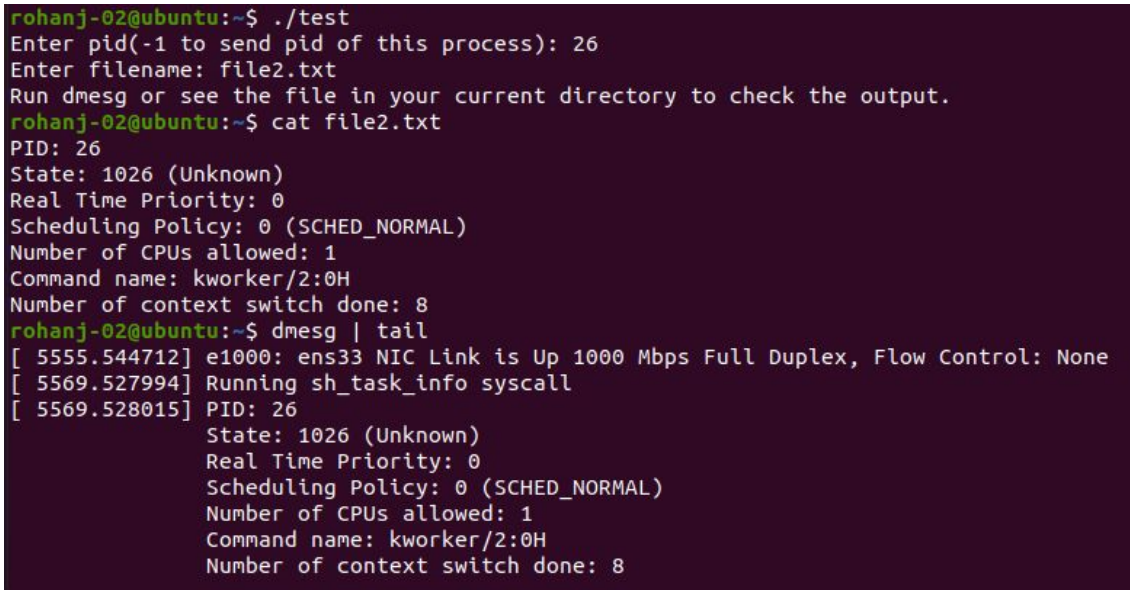
ESRCH
ENAMETOOLONG
ENOENT
EIO
The conditions for their return are explained above.

Test Cases

Sample Output 1:

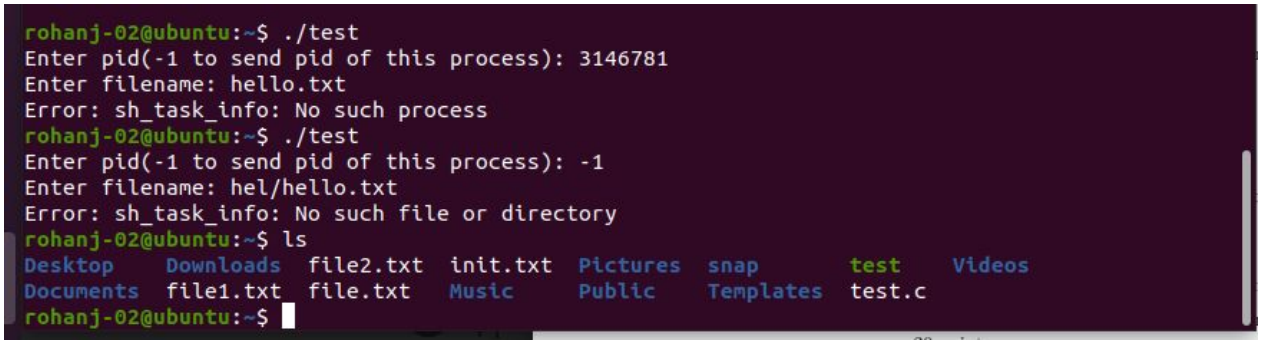


Sample Output 2:



Error Handling

To test `ESRCH`, send an invalid pid like a negative number or a really large number.
To test `ENOENT`, send a filename such as hello/hello.txt where hello/ directory does not exist.



Running test.c

To ensure giving a valid pid and checking the values, run `top` in another terminal tab and compare the values from there.