Rupanshoo Saxena | 2019096 | CSE

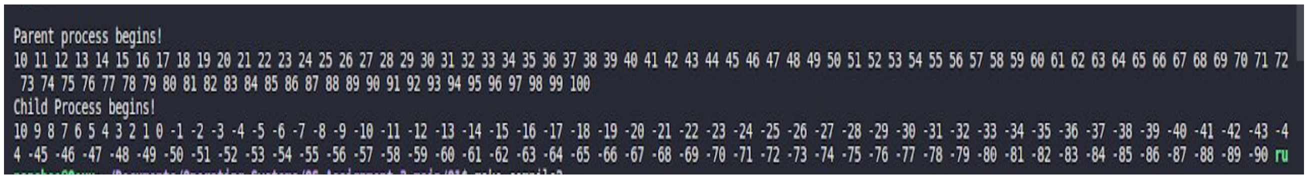
Operating Systems - Assignment 2
Date: 12th October 2020

Q1.

Case 1: using fork() → Fork is a new process which is a copy of the parent process but has its own memory and a different process ID. In this case, the parent and the forked process, both process the same code, but they execute the same code independently from each other.

When we call fork() in our code, a copy of all the pages in the parent process gets created and stored in the forked process's independent memory. So, when the child process runs, it accesses the global variable which has been stored in its independent memory and since this variable hasn't been altered yet and still stores the value 10, the numbers are decremented and printed from 10 to -90.

When the parent process is executed, it accesses the global variable stored in its own designated memory which has nothing to do with the copy of the same variable which was stored in the fork process memory, hence the numbers are generated from 10 to 100. Change in the value of the global variable stored in the forked process's memory doesn't change the value of the global variable stored in the parent process's designated memory.

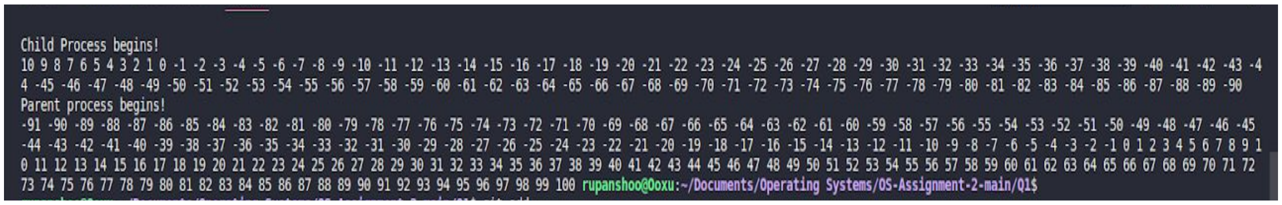


```
Parent process begins!
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Child Process begins!
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -4
4 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -88 -89 -90 ru
```

Case 2: using threads → Threads are also new processes that are created. A process can have multiple threads that execute in parallel and they share the same code and the same memory as the parent thread.

In the submitted code, the main is a process that creates a new thread called child. Main and child both share the same set of code and the same memory. Hence, when the child thread's execution completes, the value of the global variable at that moment is set at -90. And since both the child and parent thread share the same memory, when the parent thread gets executed after the child thread, at the beginning of its execution, the value of the global variable is set at -90 and not 10. Hence the parent thread prints the values of the global variable from -90 to 100.

This is the difference that is observed in the two codes - using fork and using threads. Fork makes a copy of the variable for the forked process which is independent of the parent process whereas in threads, the parent and child thread share the same memory.



```
Child Process begins!
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -4
4 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -88 -89 -90
Parent process begins!
-91 -90 -89 -88 -87 -86 -85 -84 -83 -82 -81 -80 -79 -78 -77 -76 -75 -74 -73 -72 -71 -70 -69 -68 -67 -66 -65 -64 -63 -62 -61 -60 -59 -58 -57 -56 -55 -54 -53 -52 -51 -50 -49 -48 -47 -46 -45
-44 -43 -42 -41 -40 -39 -38 -37 -36 -35 -34 -33 -32 -31 -30 -29 -28 -27 -26 -25 -24 -23 -22 -21 -20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 1
0 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 rupanshoo@Ooxu:~/Documents/Operating Systems/OS-Assignment-2-main/Q1$
```

Links used for reference: https://www.youtube.com/watch?v=nVESQQg-Oiw

Q2.

Function Implementation Steps:

1. Adding the function to syscall tables.
2. Created a separate folder called sh_task_info which contains 2 files → a .c file which contains the code for the syscall and a makefile.
3. Made changes to the main Makefile in the parent kernel folder.

Code Implementation:

- The implementation of the sh_task_info syscall has been done in the sh_task_info folder, in the sh_task_info.c file.
- I have taken **2 arguments** in the function - long PID and char* fileName.
- **Long PID** stores the pid of the process which we have to find using task_struct and **char * fileName** is the name of the file where we have to write the details of the process whose PID has also been passed as an argument.
- A local copy of the filename is created in buffer.
- **find_get_pid()** is used to get pid_struct which can be used to get_task_struct of the process whose PID has been passed.
- **If the task_struct task we get is null,** that means that the process is not running. This error handling is done using **EFAULT**. If task_struct task is not null that means that the process is running.
- Here, if the process is running, we print the details of the process in the kernel using **printk()**.
- Next we open a file in write or create mode, the name of the file is the one which has been passed to the syscall as a parameter by the user. The file is opened using **filp_open()**.

- Next, we also **check whether the file is able to open** or not. If the file doesn't open then I have also handled this error → the function returns EFAULT and exits.
- If the file is able to open, then we further write the details of the process to this file using **kernel_write()** function.
- kernel_write() requires an array which it prints and I saved the values of the process in the array using **strcat** and **strcpy.**
- After the writing process is done, the file closes using **filp_close()** and the syscall **returns 0, indicating successful implementation.**
- The test.c file prints **"works"** if the syscall returns 0, else it prints **"doesn't work"**.

We can get a list of the on-going processes with their details and IDs using the following instruction:

gnome-system-monitor

The sequence of commands used to compile and check the kernel everytime were:

1. cd Changed-linux-5.9.1 (i.e. cd Name_of_kernel_directory)
2. sudo su (to go in the root terminal)
3. yes '' | make localmodconfig
4. make -j2
5. make modules_install install
6. reboot

The messages that are being printed in the kernel can be checked using the command

dmesg

Links used for reference:
https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
https://stackoverflow.com/questions/59306780/linux-kernel-module-problem-with-kernel-write-function
https://stackoverflow.com/questions/56531880/how-does-the-kernel-use-task-struct