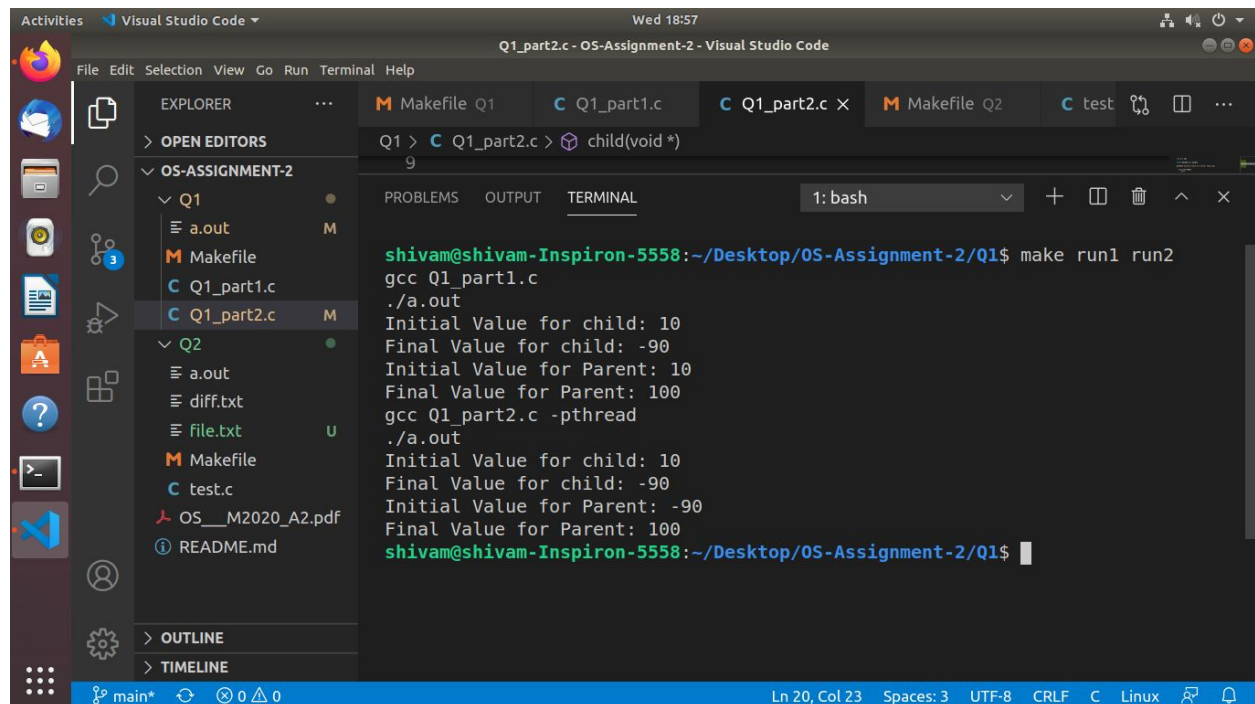


First Part:

Output:



```
shivam@shivam-Inspiron-5558:~/Desktop/OS-Assignment-2/Q1$ make run1 run2
gcc Q1_part1.c
./a.out
Initial Value for child: 10
Final Value for child: -90
Initial Value for Parent: 10
Final Value for Parent: 100
gcc Q1_part2.c -pthread
./a.out
Initial Value for child: 10
Final Value for child: -90
Initial Value for Parent: -90
Final Value for Parent: 100
shivam@shivam-Inspiron-5558:~/Desktop/OS-Assignment-2/Q1$
```

- To get output, run this on your terminal: **make run1 run2**
- The key difference between process and threads that can be seen in this question solution is that A process is an isolated execution means it does not shares the memory with any other process unit, whereas a thread is not isolated and shares memory.

Let's look at the output:

- In case of the process(Q1_part1): The initial values of parent and child are both 10, the way it is meant to be because processes is an isolated execution and don't share a memory that's why a copy of the global variable has been made, i.e operations of child process have been done on one copy of the global variable and the operations of parent are done on another copy. In the child process made by fork, it didn't share the memory(or we can say the global variable), that's why the initial values of parent and child are same, the operations done on a global variable in the child was not reflected in the parent process when we tried printing the value of a global variable.
- Incase of threads(Q1_part2) as we discussed they share memory and hence the initial values of child and parent are different(10 and -90). All the operations are done only on that global variable no copy of it was made. The child process and the parent shared the same memory (i.e, the global variable here). Hence the operations done on global

variable were reflected in the parent process when we printed the value of the global variable.

(I have used the word “Copy” just to make things understandable.)

Second Question:

- **To get output, run this on your terminal: make run**
- Description of code: First, we will see the test.c file, in the test.c file I have extracted arguments from argv, i.e user has to pass the arguments while writing ./a.out on the terminal like ./a.out 907 name.txt. Now the system call I have called is at 548 no. in syscall_64.tbl file. So, in test.c file I passed 548 in syscall() to tell the program which system call I want to use. The second argument of syscall(which is the first argument of the system call sh_task_info that I have defined in /usr/src/linux-5.9/kernel/sys.c, first argument is pid(type: pid_t) and filename(type: char*). In test.c, the two arguments that I'm taking from the user are just these two and then pass it in the syscall function like this: syscall(548,pid, filename).
- Explanation of system call sh_task_info defined in /usr/src/linux-5.9/kernel/sys.c, firstly I defined a task_struct pointer named task, I got the task by find_task_by_vpid() function by passing the pid value(**one of my argument**) in this, because we were supposed to print the details of the process corresponding to this pid. Then I started copying the fields of the task in one string array. Some of the fields of task were of long type, so I used sprintf to change their format to “%s” which is string so that they can be concatenated to the string. I used filp_open() (**and made it equal a struct file* pointer named file, for ex. file = filp_open(.....,.....);**) to open a file, and kernel_write to write in the file and filp_close() to close the file. At last

after concatenating all the fields of task in that string array, i wrote that string array in the file using `kernel_write` and then closed the file. I also used `strncpy_from_user` to copy the filename passed to the system call as an argument to a string. This was to ensure that the filename doesn't get distorted when accessed in kernel space.

Inputs

- In the makefile of part2, the user should give his arguments there, after `./a.out`, using a space after an argument. Two arguments should be given, first: pid, second: filepath. If the user doesn't know what pid value to pass, he/she can check the process list by running this command: `ps -a`, on terminal, and pass any pid value from the list.

Sample input: `./a.out 907 name.txt`

Expected Output:

- First, the user will see this message: "System call returned with, Success", only in case of successful compilation. We will talk about the errors in the later section.
- In `dmesg` and the `filename.txt` that the user has entered, the details of the process corresponding to the pid value passed by the user will be printed. `Filename.txt` will be created in the same directory as `test.c`

Errors handled:

- If the task corresponding to the pid value passed by the user doesn't exist. It will throw an error in the `dmesg` that task does not exist. Also before `dmesg`, the user will see "System call returned with No such process exist". Just try passing a pid randomly. (Result of return `-ESRCH`).
- Also, I have handled one more error that, if the `strncpy_from_user` fails it will throw an `-EFAULT` error in `dmesg`. (BAD ADDRESS

ERROR), this can be interpreted if we pass a null string in place of filename.txt.

- If the file can't be created due to permission denied error, it will throw that error too that "System call returned with, No permission".(Result of return -EACCESS. To reproduce this error: give the filepath as /home/file.txt, you will get a permission denied error.