OSL LAB-1

Team Members: Celestine Machuca.
Thishan Warnakulasooriy.

Task 1

Bash Script

```
#!/bin/bash
# GPIO pin number
gpio=136

# Export GPIO pin to userspace
echo $gpio > /sys/class/gpio/export

# Set GPIO pin as output
echo "out" > /sys/class/gpio/gpio$gpio/direction

# Toggle GPIO pin
while true; do
  echo "1" > /sys/class/gpio/gpio$gpio/value
  echo "0" > /sys/class/gpio/gpio$gpio/value
done

# Unexport GPIO pin from userspace
echo $gpio > /sys/class/gpio/unexport
```

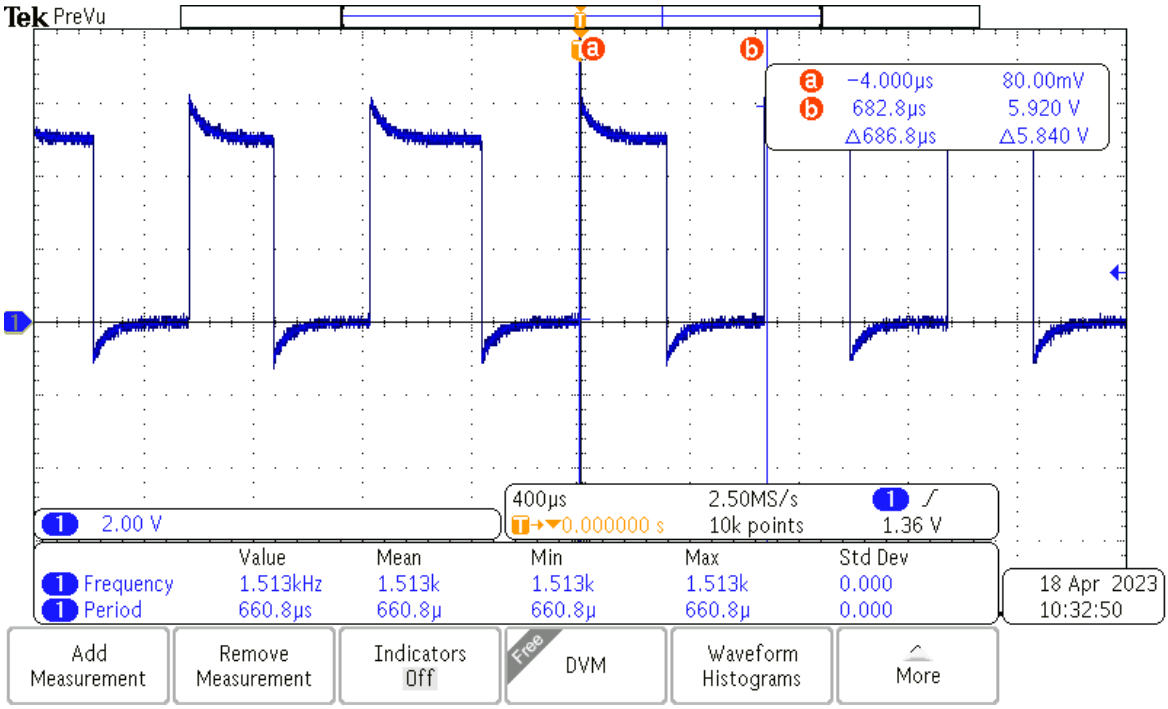

Fig 1.1 Measured Frequency 1.513 Khz

C Code

```c
#define GPIO5_START_ADDR 0x49056000
#define GPIO5_SIZE 0x00001000
#define GPIO_OE 0x034
#define GPIO_SETDATAOUT 0x094
#define GPIO_CLEARDATAOUT 0x090
#define GPIO_DATAOUT 0x03C
#define PIN 0x0100

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    volatile unsigned int *gpio_oe_addr = NULL;
    volatile unsigned int *gpio_setdataout_addr = NULL;
    volatile unsigned int *gpio_cleardataout_addr = NULL;
    volatile void *gpio_addr = NULL;

    int fd = open("/dev/mem",
                  O_RDWR | O_SYNC);
    gpio_addr = mmap(0, 4096, PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, GPIO5_START_ADDR);

    gpio_oe_addr = gpio_addr + GPIO_OE;
    gpio_cleardataout_addr = gpio_addr + GPIO_CLEARDATAOUT;
    gpio_setdataout_addr = gpio_addr + GPIO_SETDATAOUT;

    if (gpio_addr == MAP_FAILED) {
        printf("mmap failed, %x\n", gpio_addr);
        exit(1);
    }

    *gpio_oe_addr &= ~PIN;

    int i = 0;
    while (1) {
        *gpio_setdataout_addr = PIN;
        for (i = 0; i < 10; i++) continue;
        *gpio_cleardataout_addr = PIN;
    }

    close(fd);
    return 0;
}
```

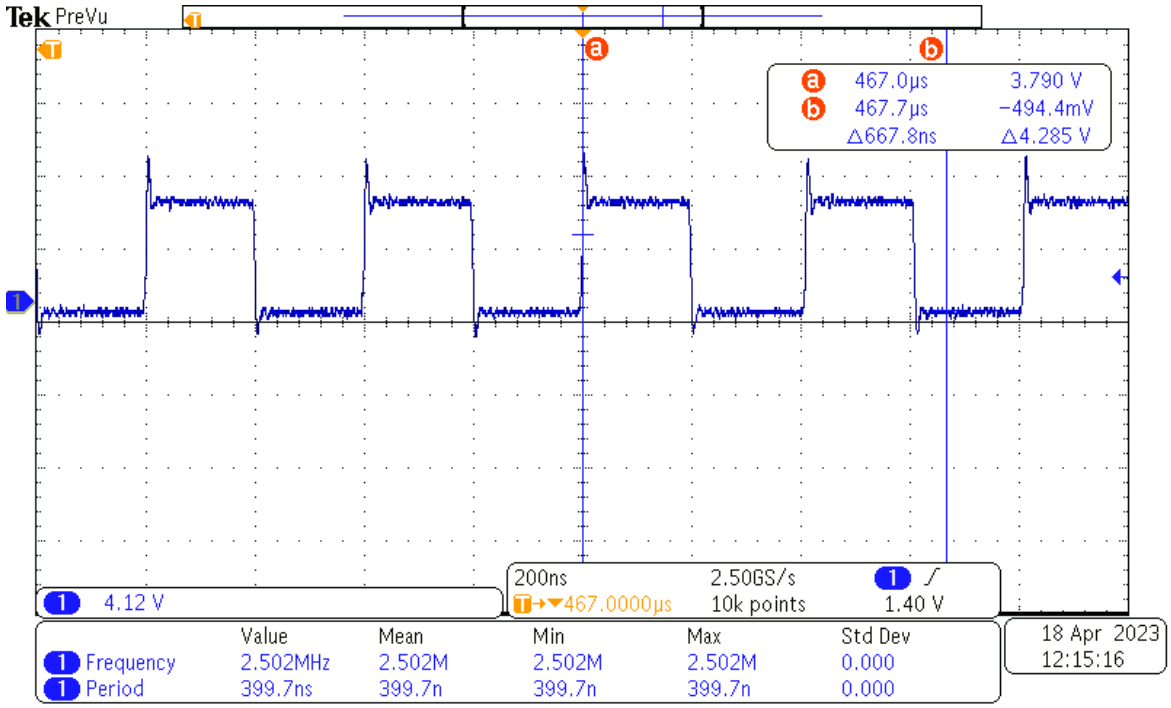

Fig 1.2 Measured Frequency 1.4974 Mhz

As we can observe from above screenshots, the direct register access approach is much faster than the shell script. This is because after memory mapping when we write directly into the shared memory space, it is immediately available and indicated as high or low in the GPIO 136 pin. When using the shell script the CPU will be performing different function calls which takes a longer duration to write into the specific GPIO pin.

Task 2

Approach 1

To create and manipulate two competing processes, we used the fork() call taught in the lectures. The idea was to create two functions that output to the screen, and track their response time (Elapsed t_exec) so as to observe how the scheduler executes them while influencing and changing the priorities(renice).

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>

#define MAX_TIME 30

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./print_string <string>\n");
        return 1;
    }
    char *niceness = argv[1];
    long long counter = 0;
    time_t start_time, current_time;
    pid_t pid = getpid();

    // Get the start time
    time(&start_time);

    // Run for 60 seconds
    do {
        counter++;
        time(&current_time);
    } while (difftime(current_time, start_time) < MAX_TIME);

    // Generate the file name
    char file_name[50];
    snprintf(file_name, sizeof(file_name), "%d.txt", pid);

    // Write the counter value to the file
    FILE *file = fopen(file_name, "w");
    if (file == NULL) {
        printf("Error: Unable to open the file.\n");
        return 1;
    }

    fprintf(file, "counter = %lld\n", counter);
    fprintf(file, "niceness = %s\n", niceness);

    fclose(file);

    printf("Counter value is written to the file %s\n", file_name);

    return 0;
}
```

The process are called by a bash script that runs the process with different niceness values.

```
# Compile the C program
gcc -o counter counter.c

# Declare an array of niceness values
niceness_values=(0 0 0 0 0 0 0 0 0 0)

# Spawn 10 instances of the program with different niceness values
for i in "${!niceness_values[@]}"; do
    nice -n "${niceness_values[$i]}" ./counter "${niceness_values[$i]}"&
done

# Wait for all instances to finish
wait

echo "All instances have finished."
```

The results are shown in the table below.

Results

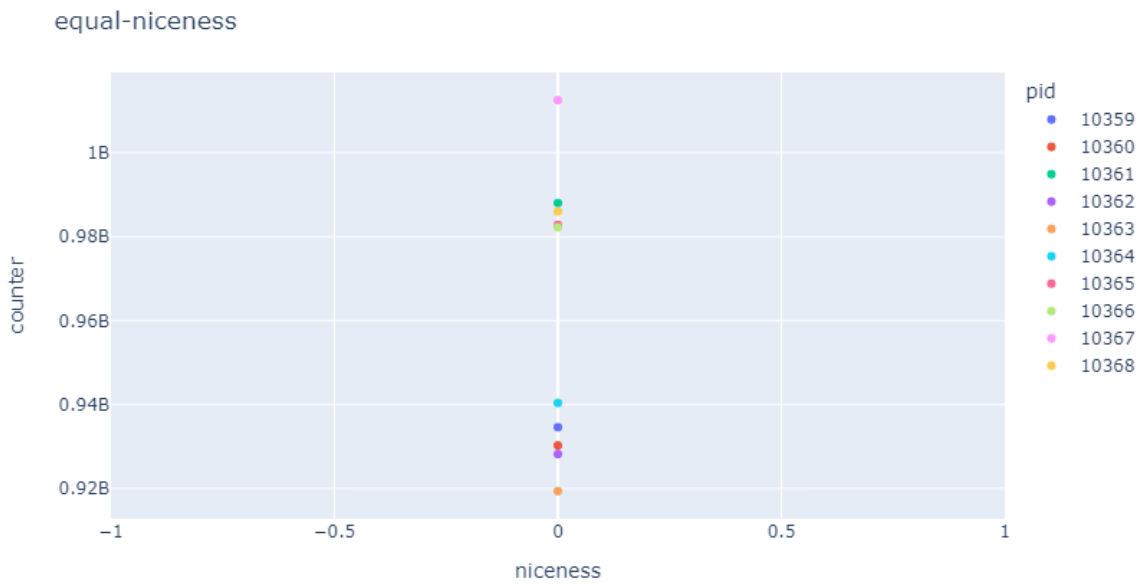

Fig 1.3 Equal Niceness Values

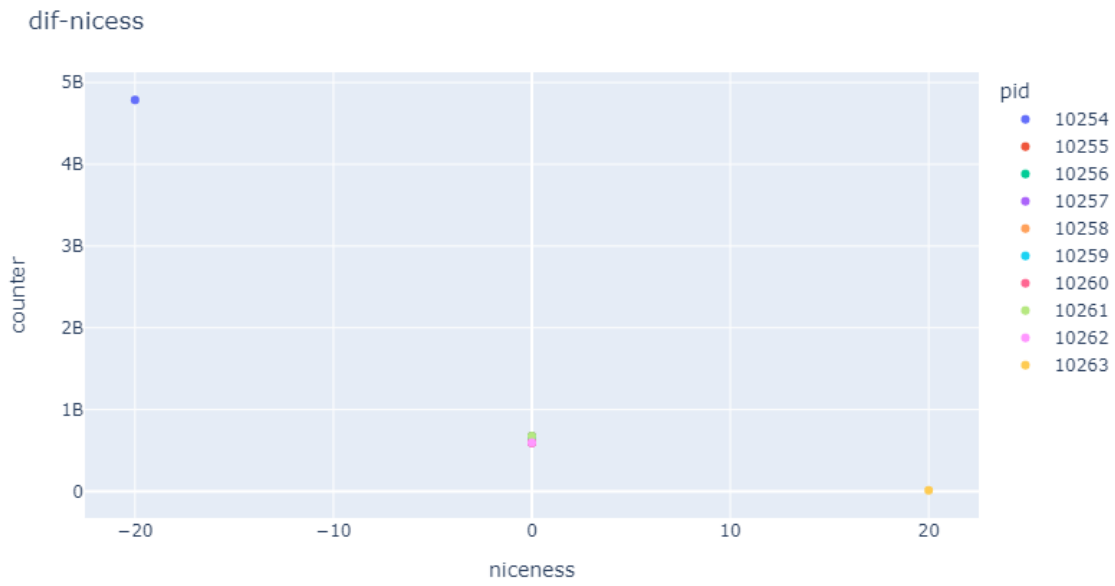

Fig 1.4 Different Niceness Values

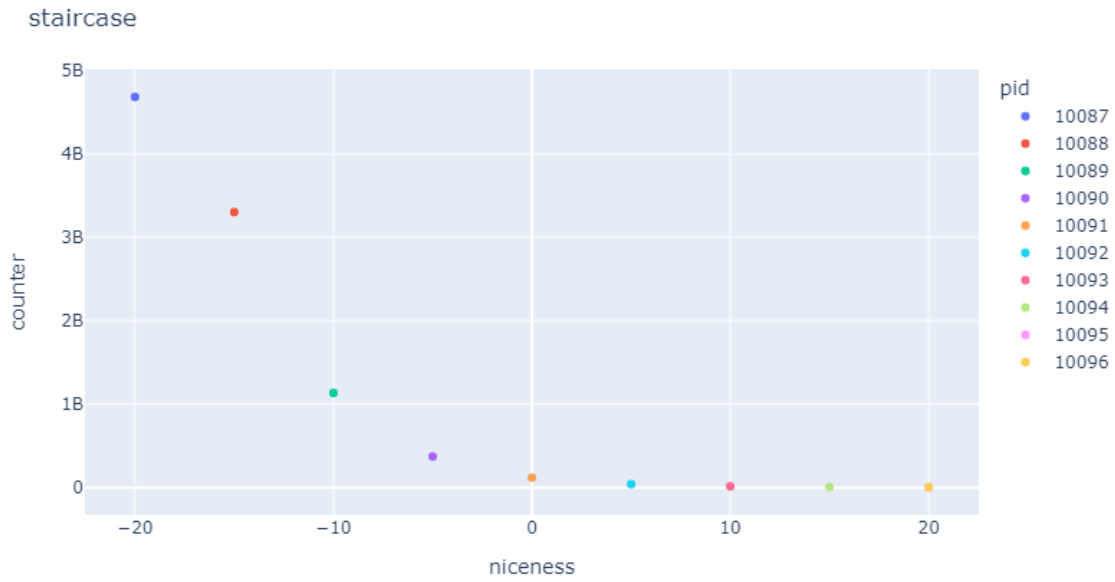

Fig 1.5 Stair Niceness Values

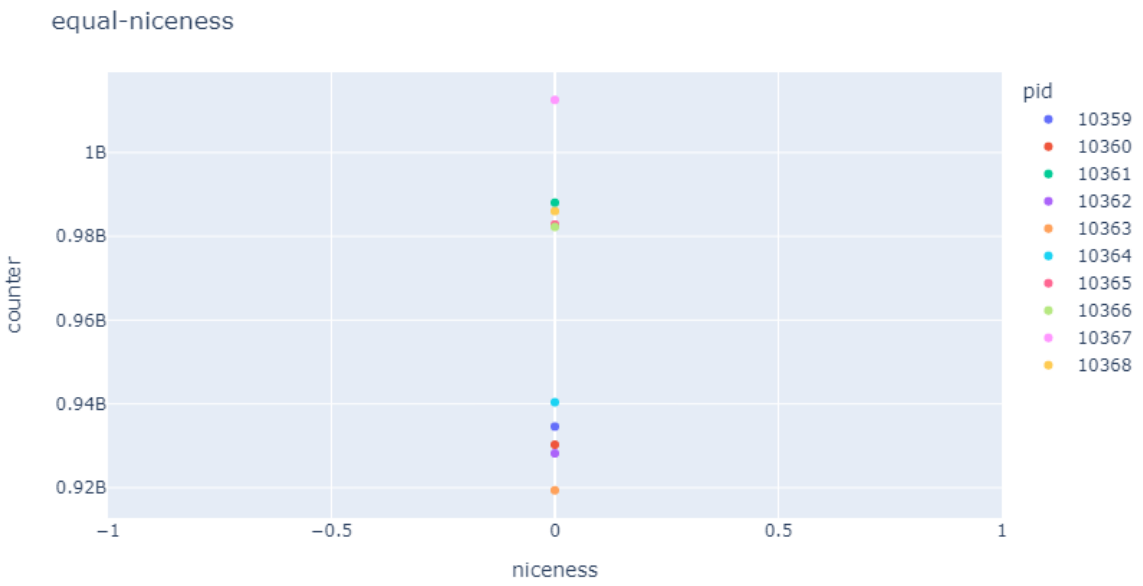

Fig 1.6 Equal Niceness Values

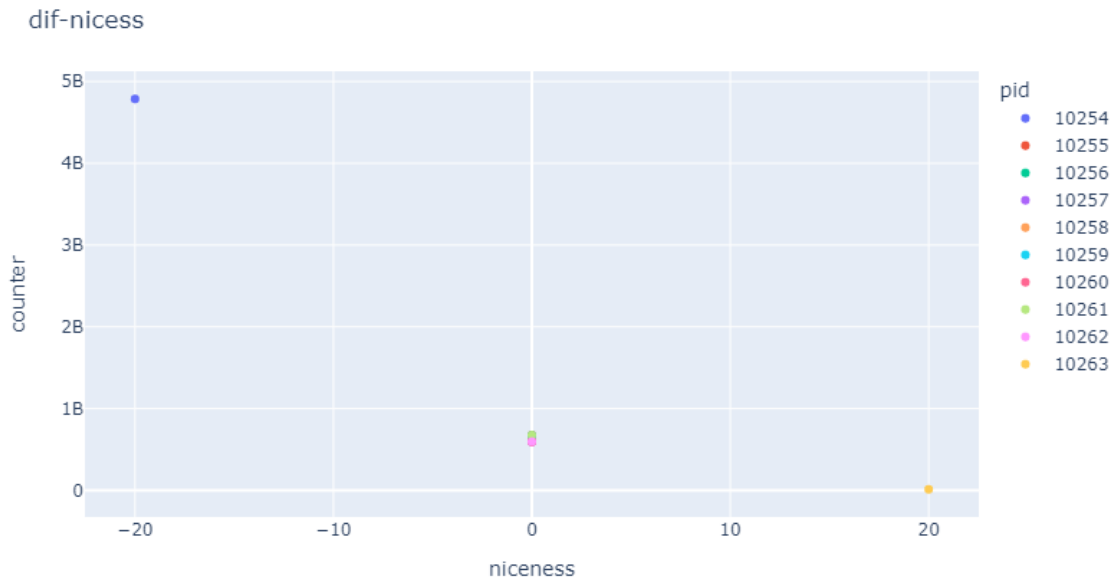

Fig 1.7 Different Niceness Values

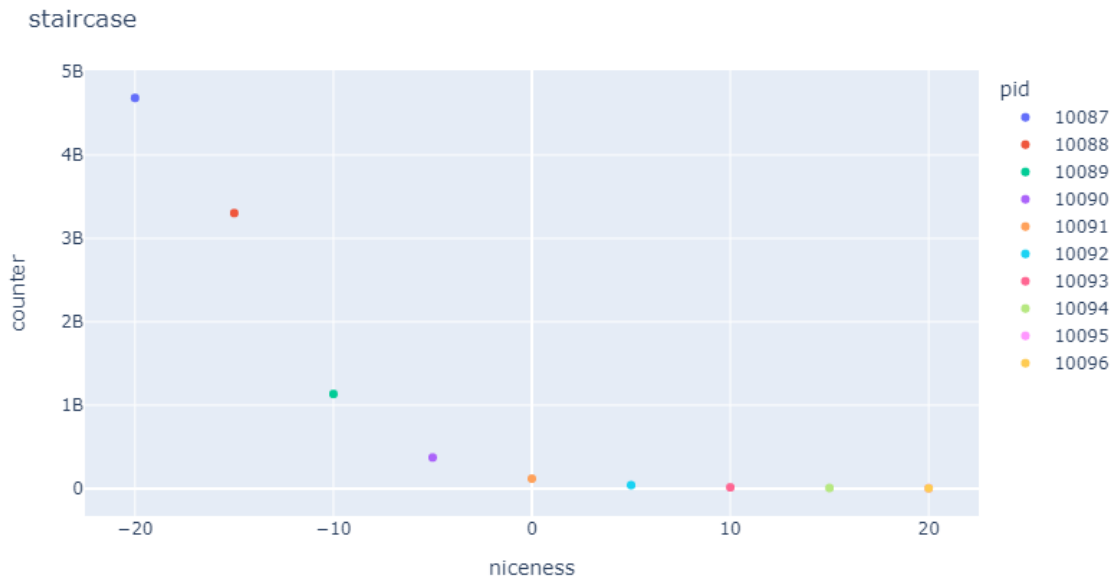

Fig 1.8 Stair Niceness Values

Sum of work done by test case

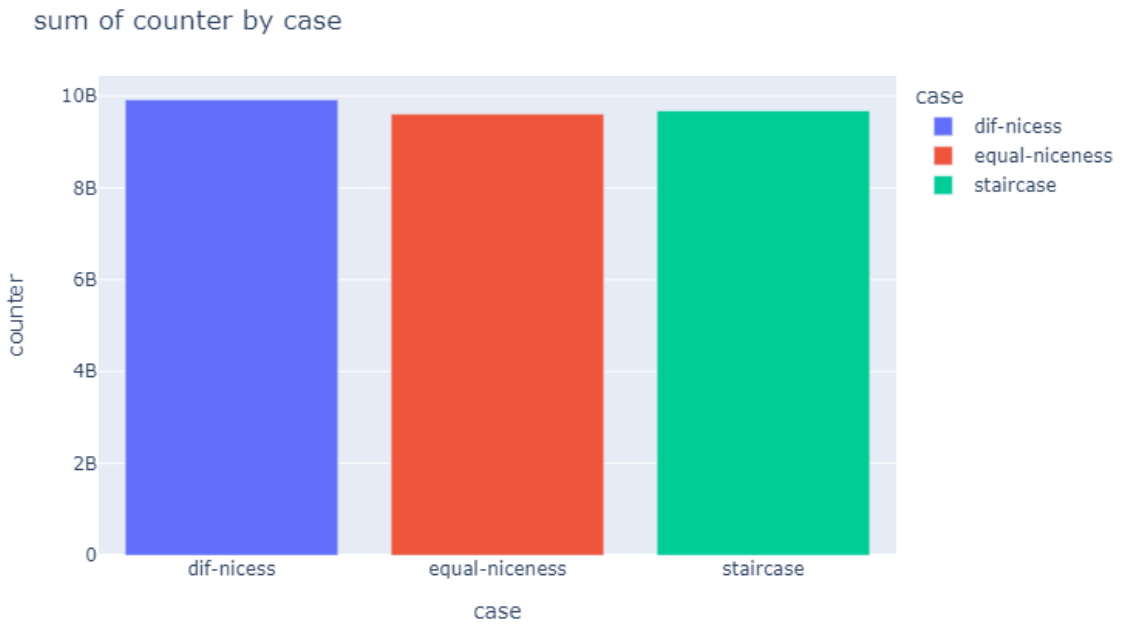

Fig 1.9 Sum of work done by test case

- Change the „nice value“ of one process by using the renice command. What’s the effect to the output?

The renice command changes the niceness value of a process. The lower the value, the higher the priority of the process. The higher the value, the lower the priority of the process. The default niceness value is 0. The renice command can be used to change the niceness value of a process.

Process tested with niceness value of 0.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/time.h>



int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    int is_child = pid == 0;

    int iterations = 1000000;
    for (int i = 0; i < iterations; i++) {
        if (is_child) {
            putchar('x');
        } else {
            putchar('.');
        }
    }
}
```

Output of the process with

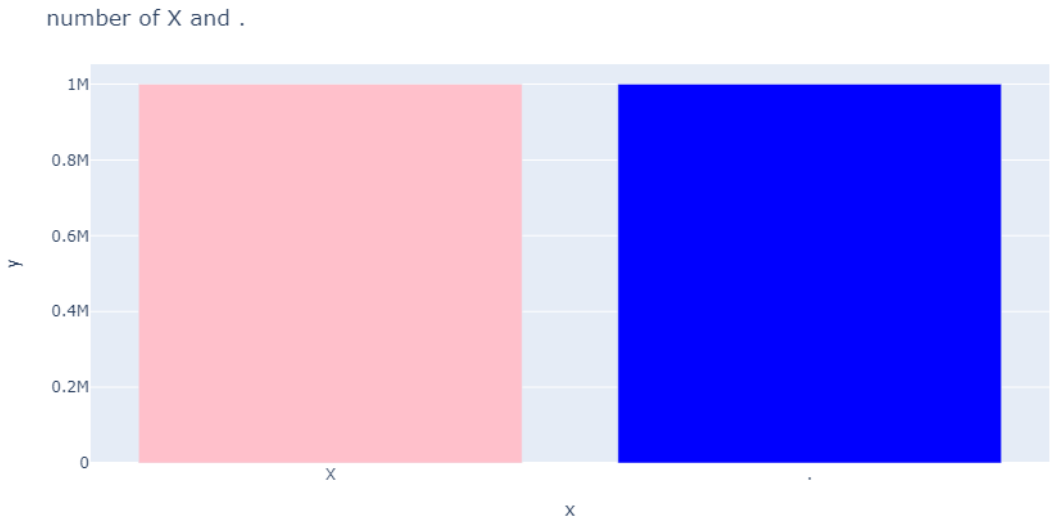

Fig 1.10 Output of the process with niceness value of 0

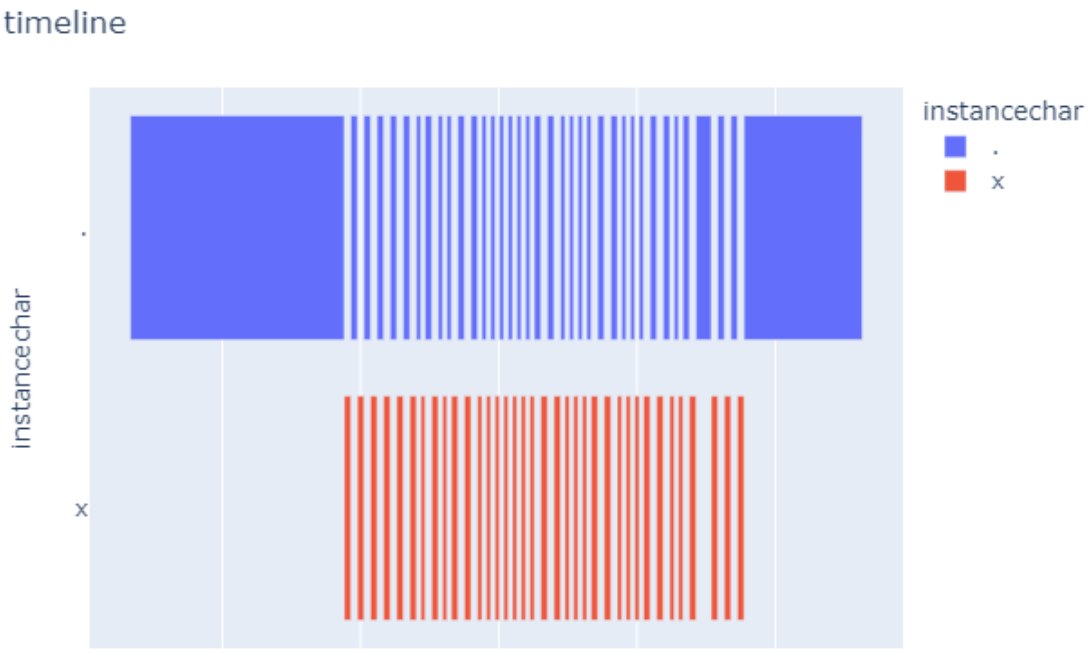

Fig 1.11 timelane of the process with niceness value of 0

- Insert a waiting period of 1 ms after each output. What's the effect to the assignment of calculation time?

Fig 1.12 Output of the process with niceness value of 0 and 1ms delay

Conclusion from Approach 1

As seen from the results, the processes with the same niceness value are executed yield about the same counter val, contrary to the case where -20 and 20 are used where there is a clear distinction between the two processes with the negative niceness value yielding a higher counter value than the positive niceness value.
A interesting side note is that the niceness did not change the total throughput of each case in a significant way, as the total work done by each case is about the same.

Aproach 2

To create and manipulate two competing processes, we used the fork() call taught in the lectures. The idea was to create two functions that output to the screen, and track their response time (Elapsed t_exec) so as to observe how the scheduler executes them while influencing and changing the priorities(renice).

2.1 Change the nice value

A c-program with the fork() call was developed to create two competing processes, both of which write to the screen without using any stream functions.

task2a.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
int main()
{
    pid_t pid = fork();
    clock_t start, endOfParent, endOfChild;
    if (pid < 0)
    {
        perror("Fork failed");
        exit(1);
    }
    while (1)
    {
        start = clock();


        if (pid > 0)
        { // fork() returns the parent process, the process ID of the child process.
            printf("I'm Parent with pid:\t%d\n",
            getpid());
            endOfParent = clock();
            printf("Elapsed t_exec -
            Parent:\t%.4f s\n", ((double)
            (endOfParent - start)) * 1e3 / CLOCKS_PER_SEC);
        }
        else if (pid == 0)
        { // fork() returns the child process, "0".
            printf("I'm Child with pid:\t%d\n",
            getpid());
            endOfChild = clock();
            printf("Elapsed t_exec - Child:\t%.4f s\n",
            ((double)(endOfChild - start)) * 1e3 / CLOCKS_PER_SEC);
        }
    }


    return 0;
}
```

To compile and then run the c code file the below commands were entered in the terminal.

```
gcc task2a.c -o task2a
./task2a
```

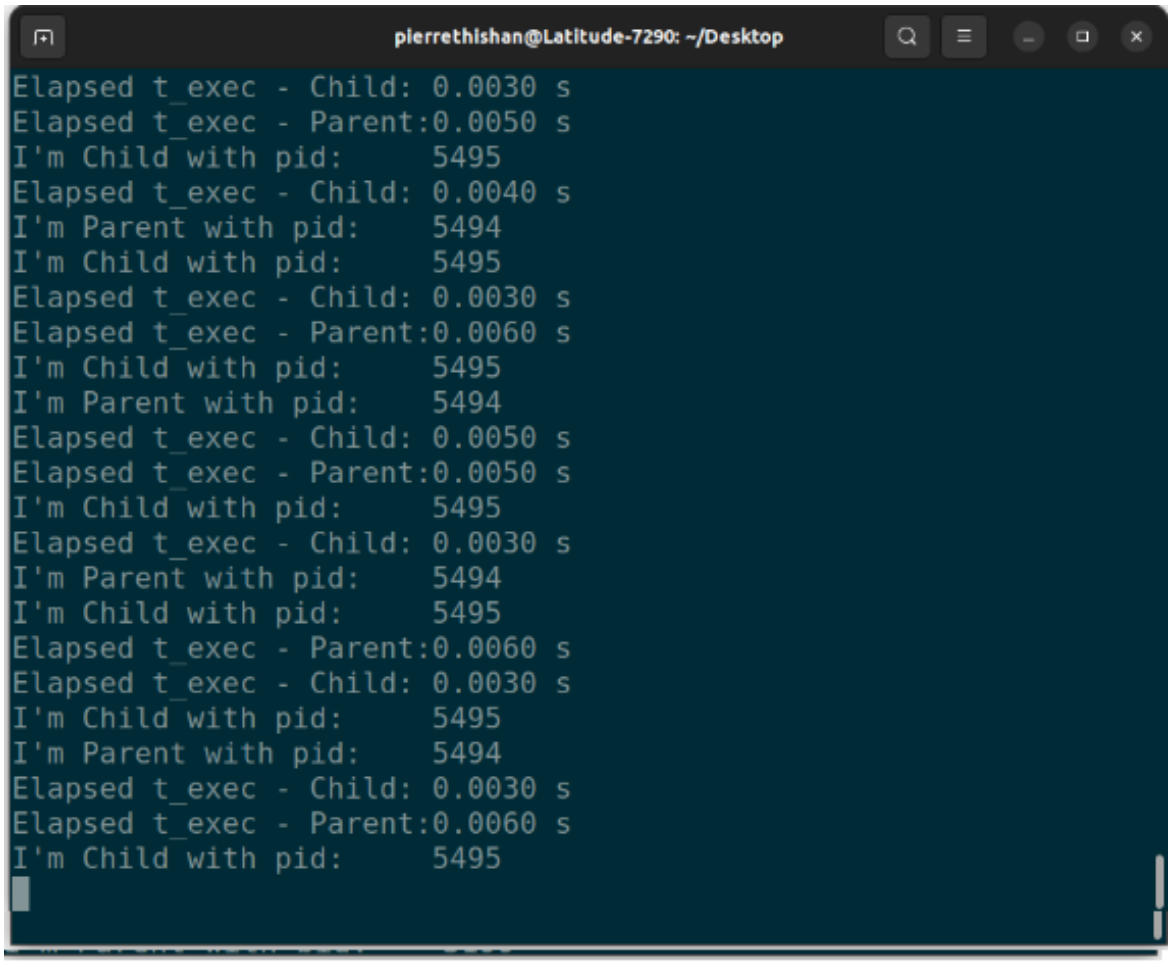

Figure 2.1 Terminal output when parent and child process's niceness values are not changed

Then we changed the nice value of the parent process to 10 using the renice command.

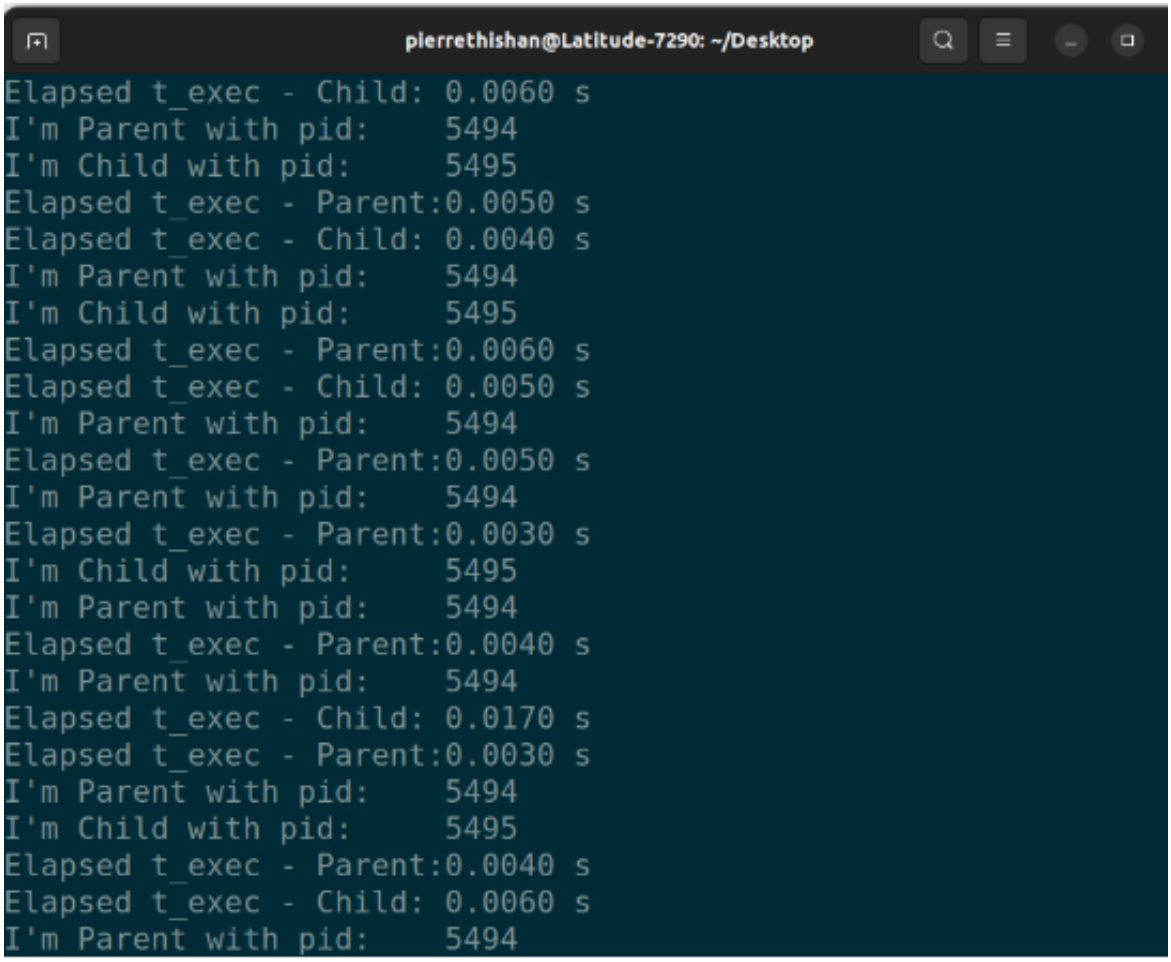

Fig 2.2 Terminal output after nice values change

As we can see above after the renice the parent process takes longer to finish executing because of the new lower priority.

2.2 The effect of a waiting period of 1 ms

usleep() was used to give a 1 ms delay after each output to the screen in the following task.

task2b.c c code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
int main()
{
    pid_t pid = fork();
    clock_t start, endOfParent, endOfChild;
    if (pid < 0)
    {
        perror("Fork failed");
        exit(1);
    }
    while (1)
    {
        start = clock();


        if (pid > 0)
        { // fork() returns the parent process, the process ID of the child process.
            printf("I'm Parent with pid:\t%d\n", getpid());
            usleep(1000);
            endOfParent = clock();
            printf("Elapsed t_exec - Parent:\t%.4f s\n", ((double)(endOfParent - start)) * 1e3 /
        }
        else if (pid == 0)
        { // fork() returns the child process, "0".
            printf("I'm Child with pid:\t%d\n", getpid());
            usleep(1000);
            endOfChild = clock();
            printf("Elapsed t_exec - Child:\t%.4f s\n", ((double)(endOfChild - start)) * 1e3 / C
        }
    }


    return 0;
}
```

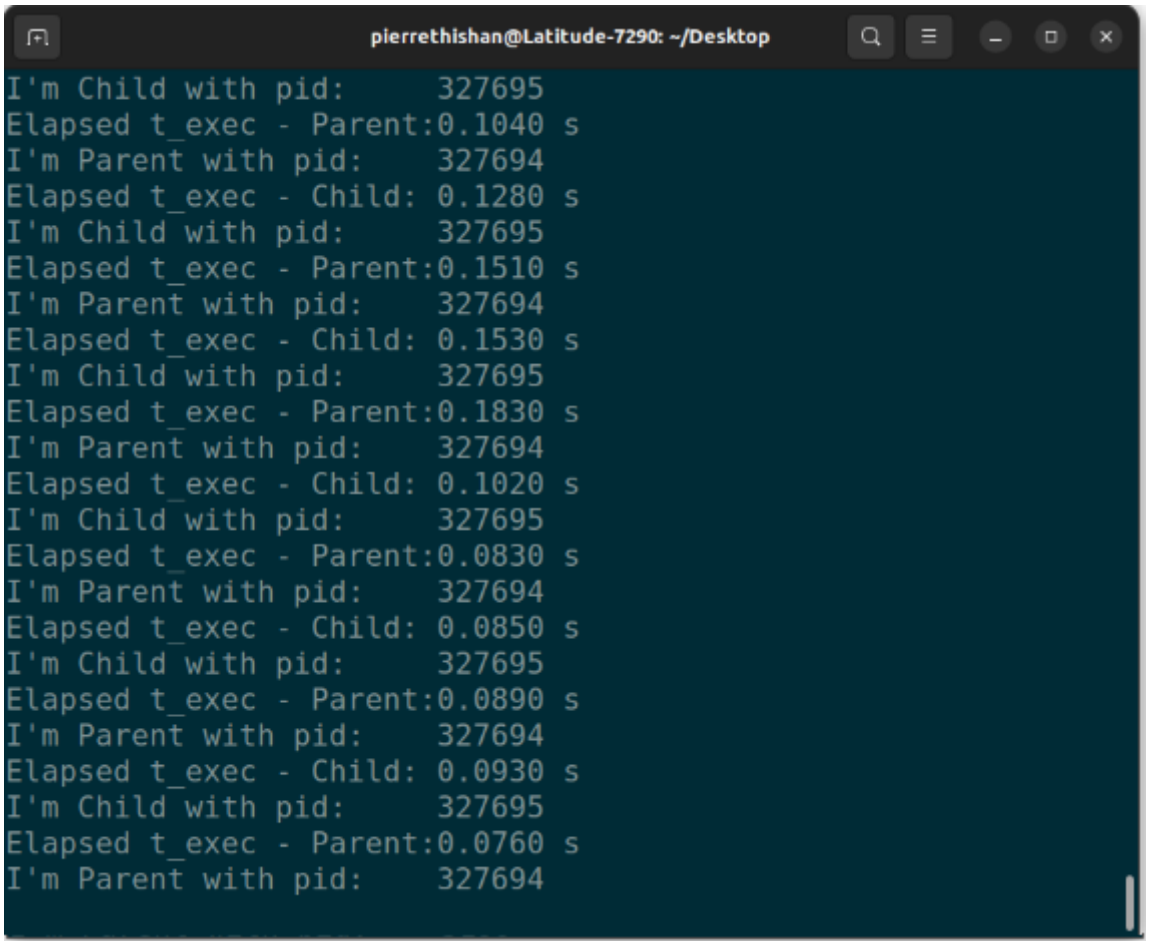

Figure 2.5 Terminal output after adding a 1ms delay

As we can observe above the execution times of each process increased with a factor of closer to 10 in comparison to before. Also, we observed that the execution times of both processes are very similar now.

2.3 Simultaneous running of time-consuming program

task 2 was run again but now while running a time-consuming program in the background. We can see that the execution times have slightly increased, the max being around 9ms (Figure 2.7).

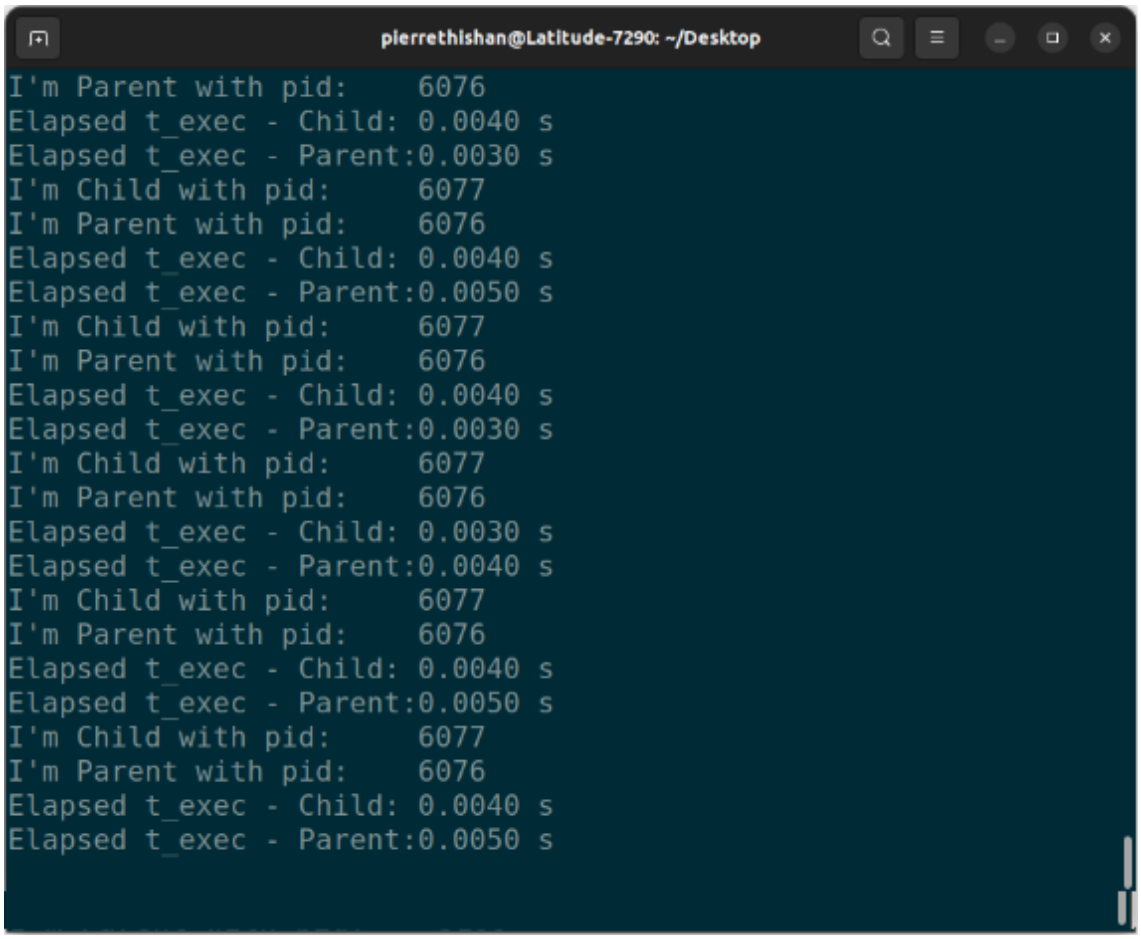

Figure 2.6 Terminal output when only task2a.c is run

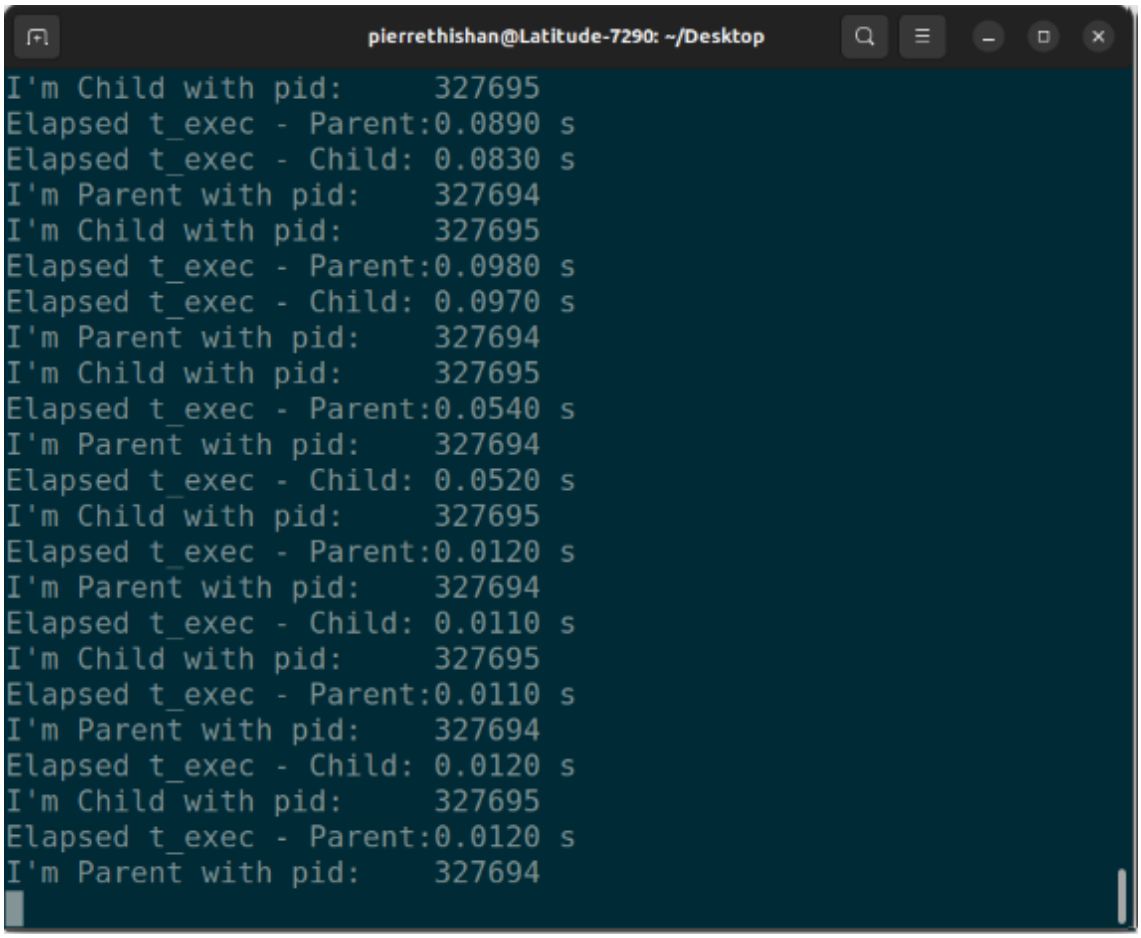

Figure 2.7 Terminal output when only task2b.c is run

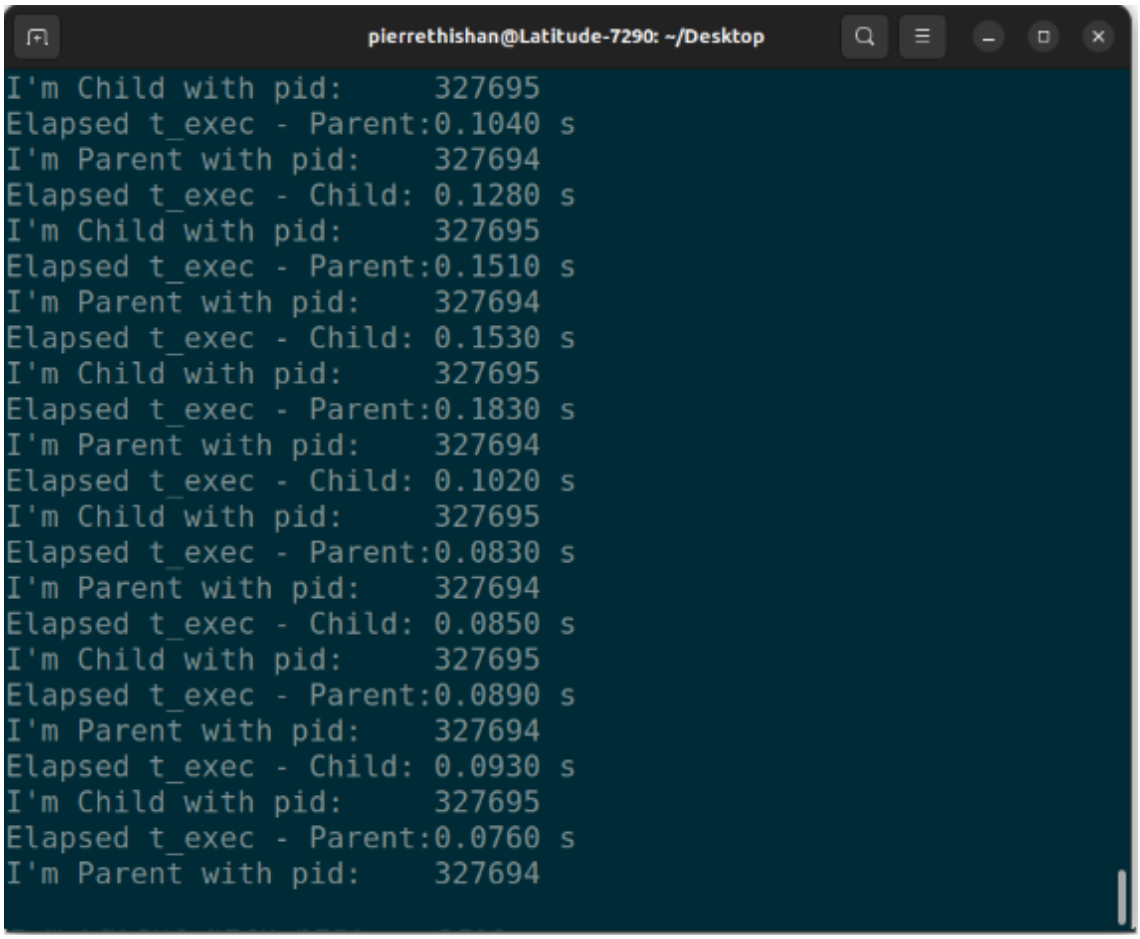

Figure 2.8 Terminal output when task2b.c and the time-consuming program are run

Used Material

- https://www.geeksforgeeks.org/nice-system-call-in-c/
- https://www.geeksforgeeks.org/renice-system-call-in-c/
- https://man7.org/linux/man-pages/man3/usleep.3.html