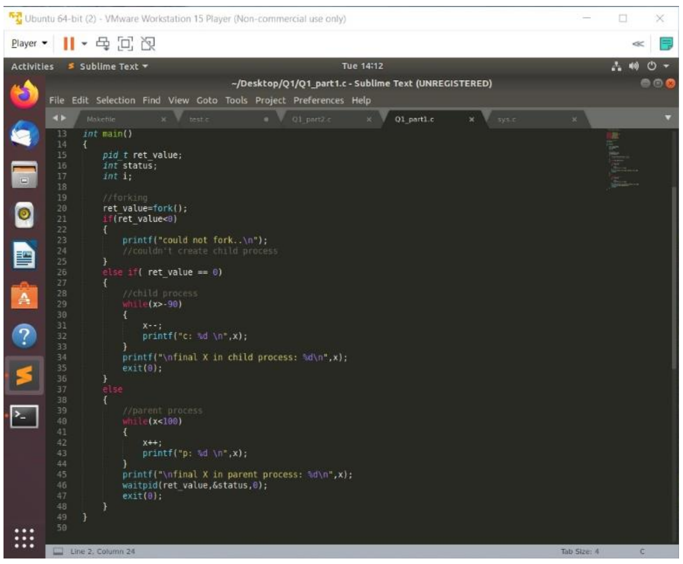
Aryan Behal OS Assignment-2 Q1 2019026

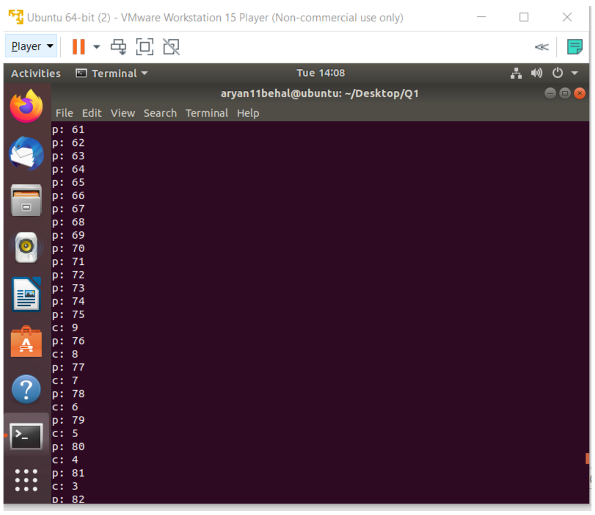
We know that for processes, complete duplication of memory takes place while in threads only partial duplication happens. Global variables are shared by threads but not by processes. The major difference in the 2 programs is based on this memory sharing concept only.

- ➔ For processes
  - For processes, I have written the following code.
  - I have simply used fork() to create a child process. In child process, I am decreasing value of global variable x up to -90 whereas increasing value of x up to 100 in my parent process. I have made my parent to wait for child termination but only at the end of parent (using waitpid()).
  - This ensures that both parent and child are executing together in the system using scheduling. (This is done to produce a similar effect as parallel processing of threads)
  - Code compiled using normal gcc.



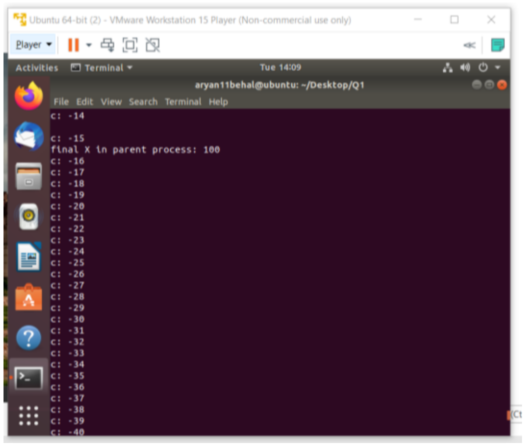
```c
int main()
{
    pid_t ret_value;
    int status;
    int i;

    //forking
    ret_value=fork();
    if(ret_value<0)
    {
        printf("could not fork..\n");
        //couldn't create child process
    }
    else if( ret_value == 0)
    {
        //child process
        while(x>-90)
        {
            x--;
            printf("c: %d \n",x);
        }
        printf("\nfinal X in child process: %d\n",x);
        exit(0);
    }
    else
    {
        //parent process
        while(x<100)
        {
            x++;
            printf("p: %d \n",x);
        }
        printf("\nfinal X in parent process: %d\n",x);
        waitpid(ret_value,&status,0);
        exit(0);
    }
}
```

As we can easily see, processor switches from parent process to child while execution and vice versa depending upon which process gets the cpu. This is completely random.



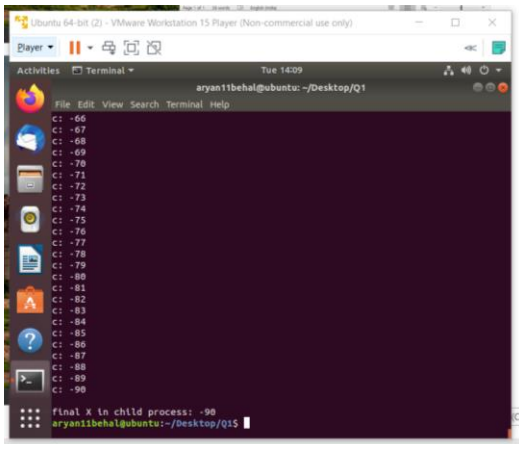
```
p: 61
p: 62
p: 63
p: 64
p: 65
p: 66
p: 67
p: 68
p: 69
p: 70
p: 71
p: 72
p: 73
p: 74
p: 75
c: 9
p: 76
c: 8
p: 77
c: 7
p: 78
c: 6
p: 79
c: 5
p: 80
c: 4
p: 81
c: 3
p: 82
```

P’s x= 75 but c’s x=9..



```
c: -14

c: -15
final X in parent process: 100
c: -16
c: -17
c: -18
c: -19
c: -20
c: -21
c: -22
c: -23
c: -24
c: -25
c: -26
c: -27
c: -28
c: -29
c: -30
c: -31
c: -32
c: -33
c: -34
c: -35
c: -36
c: -37
c: -38
c: -39
c: -40
```

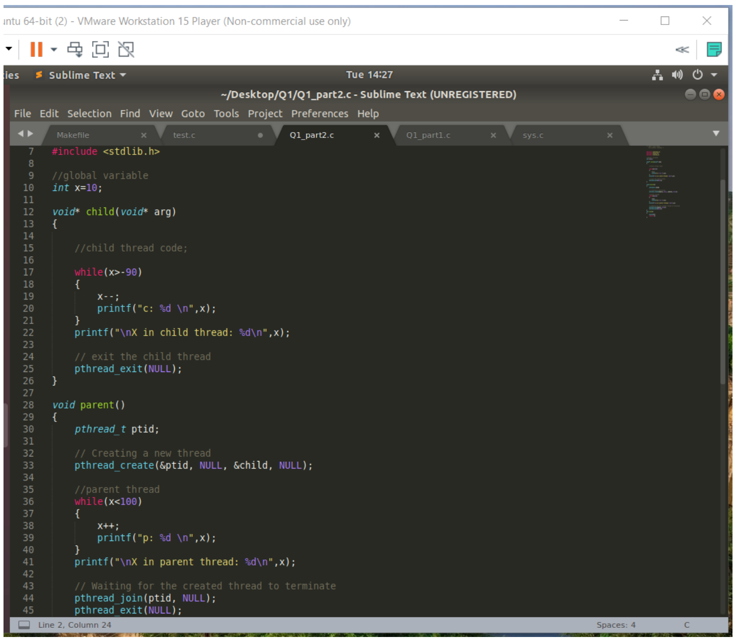
But once either of the process ends (anyone can finish early), only the other process runs. Here, parent ended and waited for child to complete its execution before exiting.



```
c: -66
c: -67
c: -68
c: -69
c: -70
c: -71
c: -72
c: -73
c: -74
c: -75
c: -76
c: -77
c: -78
c: -79
c: -80
c: -81
c: -82
c: -83
c: -84
c: -85
c: -86
c: -87
c: -88
c: -89
c: -90

final X in child process: -90
aryan11behal@ubuntu:~/Desktop/Q1$ 
```

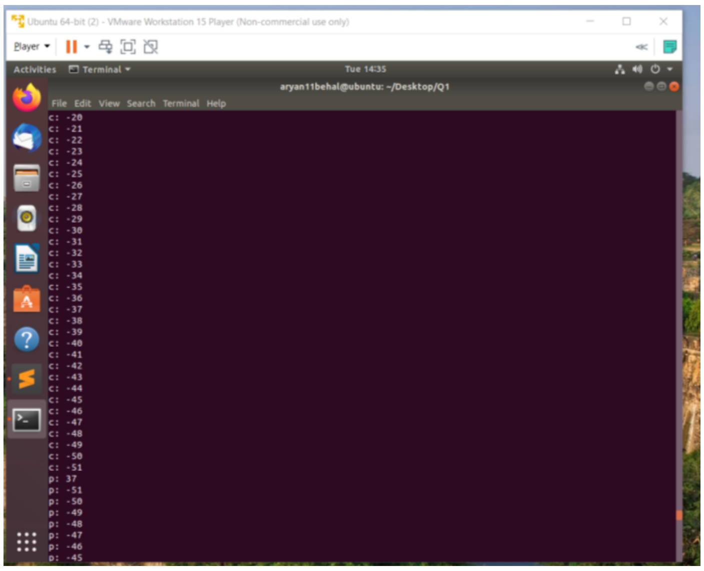
➔ For threads
   - I have written the following code.
   - I am simply using pthread_create() to create a new thread (child thread).

- Then I am increasing my value of global variable in parent thread while decreasing value in child thread.
- I am using a pthread_join() to re-join the threads made once they have completed their parallel execution and pthread_exit() to exit from thread.
- Code compiled using: **gcc Q1_part2.c -pthread -o Q1_part2**

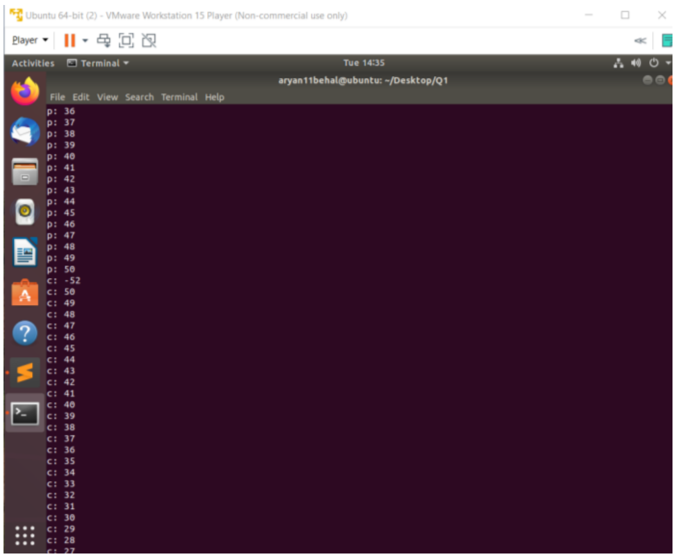


```c
#include <stdlib.h>

//global variable
int x=10;

void* child(void* arg)
{

    //child thread code;

    while(x>-90)
    {
        x--;
        printf("c: %d \n",x);
    }
    printf("\nX in child thread: %d\n",x);

    // exit the child thread
    pthread_exit(NULL);
}

void parent()
{
    pthread_t ptid;

    // Creating a new thread
    pthread_create(&ptid, NULL, &child, NULL);

    //parent thread
    while(x<100)
    {
        x++;
        printf("p: %d \n",x);
    }
    printf("\nX in parent thread: %d\n",x);

    // Waiting for the created thread to terminate
    pthread_join(ptid, NULL);
    pthread_exit(NULL);
```

Here, again as the 2 threads are running in parallel, either can get the cpu and execute itself. This order is completely random.

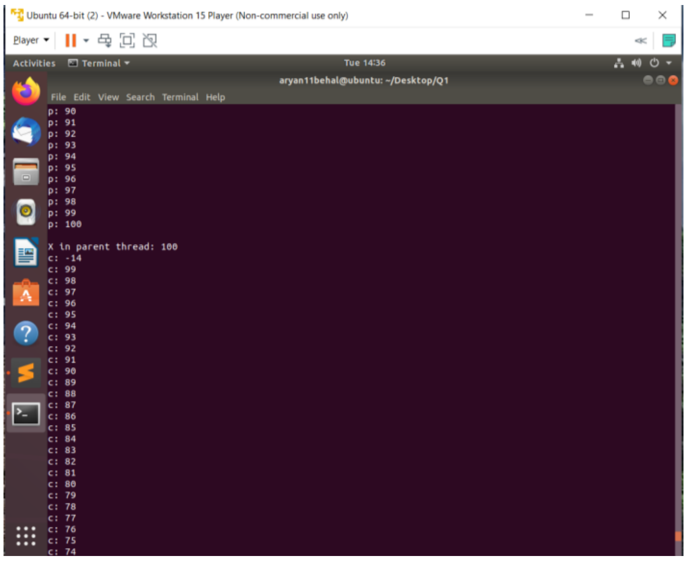


```
c: -20
c: -21
c: -22
c: -23
c: -24
c: -25
c: -26
c: -27
c: -28
c: -29
c: -30
c: -31
c: -32
c: -33
c: -34
c: -35
c: -36
c: -37
c: -38
c: -39
c: -40
c: -41
c: -42
c: -43
c: -44
c: -45
c: -46
c: -47
c: -48
c: -49
c: -50
c: -51
p: 37
p: -51
p: -50
p: -49
p: -48
p: -47
p: -46
p: -45
```



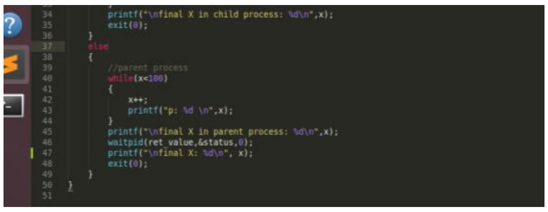
```
p: 36
p: 37
p: 38
p: 39
p: 40
p: 41
p: 42
p: 43
p: 44
p: 45
p: 46
p: 47
p: 48
p: 49
p: 50
c: -52
c: 50
c: 49
c: 48
c: 47
c: 46
c: 45
c: 44
c: 43
c: 42
c: 41
c: 40
c: 39
c: 38
c: 37
c: 36
c: 35
c: 34
c: 33
c: 32
c: 31
c: 30
c: 29
c: 28
c: 27
```

Once a thread reaches its criterion, we exit from that thread using pthread_exit(). So once the parent thread makes x=100 or child makes x=-90, we exit from them and their execution stops and that threads waits for other thread to complete execution before joining together.

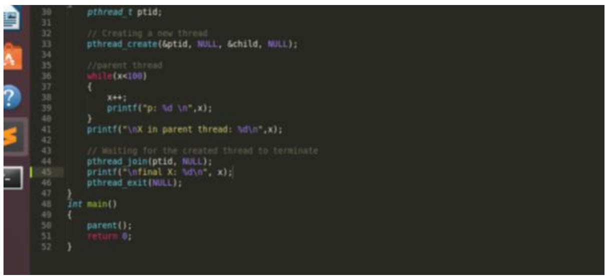


```
p: 90
p: 91
p: 92
p: 93
p: 94
p: 95
p: 96
p: 97
p: 98
p: 99
p: 100

X in parent thread: 100
c: -14
c: 99
c: 98
c: 97
c: 96
c: 95
c: 94
c: 93
c: 92
c: 91
c: 90
c: 89
c: 88
c: 87
c: 86
c: 85
c: 84
c: 83
c: 82
c: 81
c: 80
c: 79
c: 78
c: 77
c: 76
c: 75
c: 74
```

- ➔ Final value for x where parent’s and child’s executions stop will be same, 100 and -90 respectively as our conditions are such. But the 2 methods clearly have significant differences.



```
        printf("\nfinal X in child process: %d\n",x);
        exit(0);
    }
    else
    {
        //parent process
        while(x<100)
        {
            x++;
            printf("p: %d \n",x);
        }
        printf("\nfinal X in parent process: %d\n",x);
        waitpid(ret_value,&status,0);
        printf("\nfinal X: %d\n", x);
        exit(0);
    }
}
```



```c
    pthread_t ptid;

    // Creating a new thread
    pthread_create(&ptid, NULL, &child, NULL);

    //parent thread
    while(x<100)
    {
        x++;
        printf("p: %d \n",x);
    }
    printf("\nX in parent thread: %d\n",x);

    // Waiting for the created thread to terminate
    pthread_join(ptid, NULL);
    printf("\nfinal X: %d\n", x);
    pthread_exit(NULL);
}
int main()
{
    parent();
    return 0;
}
```

- > For processes, when we print x after waitpid(), it will always show 100 no matter what was the order of execution or which process finished first. But for threads, after pthread_join(), value of x depends on which thread finished later, if parent finished execution later then x=100, else x=-90. This is because global variable is shared by threads and not by processes. In process, after waitpid(), we are using parent's x copy which is 100, but the child process's copy of x is finished with the process. But for threads, x is shared and its value is set by the thread terminating later. After pthread_join(), x has value which last executed thread gave. So, it is variable.
- > In processes, when both child and parent processes start execution first time, both start with x=10 but in threads the later thread starts its value with what the former thread has made it already.
- > For processes, when processor switches to the other process, we can easily see that values are not consecutive. There is a clear jump in value of x which is not the case for threads. Threads change value from the last printed value on console but processes change value consecutive to the last value that the same process printed on console which clearly shows that processes have their own copy of global variables but not threads. (For threads, we are seeing 1 previous value from its last execution value because the reading/writing to memory takes time. So by the time, x++/x—is written to the memory and accessed for printf, the processor has already switched to other thread and it is this missed print statement that is executed.)
- > As memory is shared in threads, x can be randomly increased or decreased by parent and child. Hence the terminating conditions take longer to reach and might take infinite time too whereas in processes, for given values, parent will terminate after 90 iteration of "while" loop and child in 100 iterations of "while" loop.