Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Operating Systems-ENCS3390

Programming Task #1

Name :	Qusay Taradeh
ID No. :	1212508
Instructor :	Bashar Tahayna
Section No. :	4

Summary of Processes, Joinable Threads , Detached Threads and Parallelism

Processes

A process is an instance of a program that is being executed. Each process has an execution time and performs a specific part of the task. Therefore, we use processes to reduce the overall time required to complete the task. Every process has a parent process that holds its child process ID. We create processes, known as "child processes," using the fork function, where each child process requires one fork. Finally, the parent process must wait for its child process to finish its part of the task before the entire task is considered complete. However, if the parent process does not wait for its child process, the task will not be completed as expected.

Threads

A thread is a lightweight process that shares resources with other threads to perform a given task. It can operate concurrently or in parallel. There are two types of threads: joinable threads and detached threads.

Joinable Threads

A joinable thread is a thread that can be joined by another thread. When a thread joins another thread, it waits for the second thread to complete execution before proceeding. Joinable threads are particularly useful for tasks that require multiple parts to be completed successfully before the overall task can be considered finished.

Detached Threads

A detached thread is a thread that cannot be joined by another thread. When a detached thread terminates, its resources are automatically released back to the system. Detached threads are typically used when the parent thread does not need to wait for the child thread to complete its task.

Parallelism

The concept of parallelism is effectively applied in matrix multiplication, allowing us to divide the task into smaller parts that can be executed independently using processes or threads. This approach significantly reduces the overall execution time compared to a sequential execution. Multiprocessing and multithreading serve as practical implementations of parallelism, enabling efficient matrix multiplication.

Native Case:

There is in two pictures the code of native C code to initializing the matrices and the time variables that calculate the time for matrix multiplication of matrices:

OS_1212508 > main.c > main(int, char * [])

```c
            k++;
        }
    }

    printf("\nID * Year Matrix\n");
    k = 0;
    for (i = 0; i < N; i++) // loop to fill ID * Year matrix
    {
        for (j = 0; j < N; j++)
        {
            if (k > 9)
                k = 0;

            idXyearMatrix[i][j] = idXyear[k];
            printf("%d ", idXyearMatrix[i][j]);
            k++;
        }
    }

    printf("\nResult Matrix\n");
    start = clock(); // start measuring time
    for (i = 0; i < N; i++) // loop for multiplication as a native without any process or thread
    {
        for (j = 0; j < N; j++)
        {
            resultMatrix[i][j] = 0;
            for (k = 0; k < N; k++)
            {
                resultMatrix[i][j] += (idMatrix[i][k] * idXyearMatrix[k][j]);
```

OS_1212508 > main.c > main(int, char * [])

```c
#define N 100

void childProcess(int idMatrix[N][N], int idXyearMatrix[N][N], int resultMatrix[N][N]);

int main(int argc, char *argv[])
{
    clock_t start, end; // time variables
    float tm; // tm: time calculated
    pid_t pid; // pid : process id
    int pipeFD[4]; // pipeFD : pipe file discovery for IPC
    int i, j, k = 0; // variables for loops
    int id[7] = {1, 2, 1, 2, 5, 0, 8}; // my ID represented as digits in array
    int idXyear[10] = {2, 4, 2, 8, 6, 5, 3, 5, 2, 4}; // idXyear = id * year = 1212508 * 2003 = 2428653524
    int idMatrix[N][N]; // matrix of my ID number
    int idXyearMatrix[N][N]; // matrix of my ID multiplied by my birth year
    int resultMatrix[N][N]; // matrix wich is the result of multiplication between ID matrix and ID * year matrix

    //////// Native C code for implementing and multiplication
    printf("\nID Matrix\n");
    for (i = 0; i < N; i++) // loop to fill ID matrix
    {
        for (j = 0; j < N; j++)
        {
            if (k > 6)
                k = 0;

            idMatrix[i][j] = id[k];
            printf("%d ", idMatrix[i][j]);
            k++;
        }
```

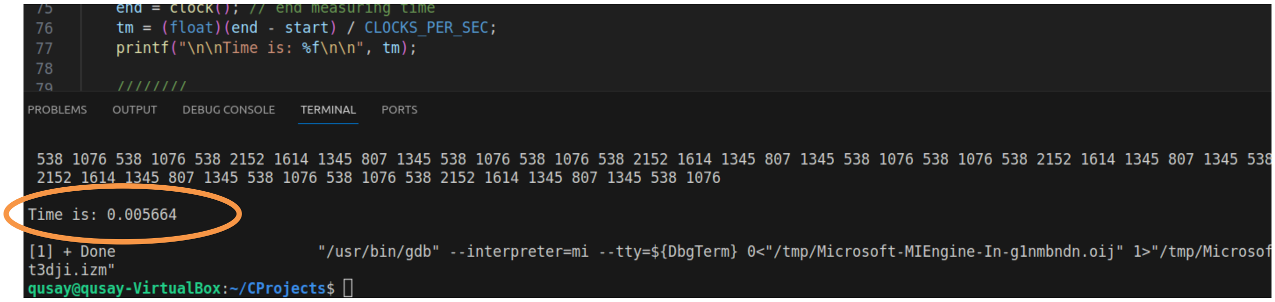
```c
        {
            if (k > 9)
                k = 0;

            idXyearMatrix[i][j] = idXyear[k];
            printf("%d ", idXyearMatrix[i][j]);
            k++;
        }
    }

    printf("\nResult Matrix\n");
    start = clock(); // start measuring time
    for (i = 0; i < N; i++) // loop for multiplication as a native without any process or thread
    {
        for (j = 0; j < N; j++)
        {
            resultMatrix[i][j] = 0;
            for (k = 0; k < N; k++)
            {
                resultMatrix[i][j] += (idMatrix[i][k] * idXyearMatrix[k][j]);
            }
            printf("%d ", resultMatrix[i][j]);
        }
    }
    end = clock(); // end measuring time
    tm = (float)(end - start) / CLOCKS_PER_SEC;
    printf("\n\nTime is: %f\n\n", tm);

    ////////
```

And below in Terminal the time that taken to do the task without process or thread which is equal 0.005664s which is 5.664ms and throughput is 176.55s^{-1}



So we notice that time complexity is O(n^3), so to solve this problem and actually in our task that multiply matrices that takes a long time specially when number of rows and cols is increased. Furthermore we go to part one of task that we will use processes to manage the execution time and make it less than native way coding the task.

Process Management:

From below snapshots we used 2 child processes to do the multiplication and we notice that the time reduced to 0.002361s which is 2.361ms instead of 5.664ms and this show the importance of multiprocesses, and the throughput equals 423.55s^{-1}

OS_1212508 > main.c > main(int, char * [])

```c
    for(i = 0; i < 2; i++)
        if (pipe(pipeFD[i]) == -1) // error handling when creating pipe failure and this return 2
        {
            printf("Can't create pipe!\n");
            return 2;
        }

    gettimeofday(&start,NULL);
    pid = fork();    // forking the process
    if (pid == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }
    if (pid == 0)
    {
        //Child process 1

        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[1][1]); //close write of process 2

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[0][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 4; } // writing and handling error

        close(pipeFD[0][1]); //close writing in process 1
        return 0;
    }

    pid_t pid2 = fork();
    if (pid2 == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
```

OS_1212508 > main.c > main(int, char * [])

```c
    pid_t pid2 = fork();
    if (pid2 == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }

    if (pid2 == 0)
    {
        //Child process 2
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[0][1]); //close write of process 1

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[1][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error

        close(pipeFD[1][1]); //close writing in process 2
        return 0;
    }
    //Parent process
    //start = clock();
    close(pipeFD[0][1]); //close write of process 1
    close(pipeFD[1][1]); //close write of process 2

    gettimeofday(&end, NULL);

    waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
    waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation

    if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1

    if(read(pipeFD[1][0], resultMatrix + N/2, sizeof(int)*N*N/2) == -1) { return 5; } // reading from process 2
```

OS_1212508 > main.c > main(int, char * [])

```c
        close(pipeFD[1][1]); //close writing in process 2
        return 0;
    }
    //Parent process
    //start = clock();
    close(pipeFD[0][1]); //close write of process 1
    close(pipeFD[1][1]); //close write of process 2

    gettimeofday(&end, NULL);

    waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
    waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation

    if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1

    if(read(pipeFD[1][0], resultMatrix + N/2, sizeof(int)*N*N/2) == -1) { return 5; } // reading from process 2

    close(pipeFD[0][0]); //close read of process 1
    close(pipeFD[1][0]); //close read of process 2
    //printMatrix(resultMatrix);


    tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
    printf("\n\nTime is: %fs\n\n", tm);

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Time is: 0.002361s

[1] + Done                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-lv2k1wtz.2rq" 1>"/tmp/Microsoft-MIEngine-Out-hhb
wyq1m.j4t"
qusay@qusay-VirtualBox:~/CProjects$
```

Now we want to increase number of child processes to 4 instead of 2 child processes and noting the difference in time and throughput.

OS_1212508 > main.c > main(int, char * [])

```c
    ///////////////end of initializing/////////
    for(i = 0; i < 4; i++)
        if (pipe(pipeFD[i]) == -1) // error handling when creating pipe failure and this return 2
        {
            printf("Can't create pipe!\n");
            return 2;
        }

    gettimeofday(&start,NULL);
    pid = fork();    // forking the process
    if (pid == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }
    if (pid == 0)
    {
        //Child process 1
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[1][1]); //close write of process 2
        close(pipeFD[2][0]); //close read of process 3
        close(pipeFD[2][1]); //close write of process 3
        close(pipeFD[3][0]); //close read of process 4
        close(pipeFD[3][1]); //close write of process 4

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[0][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 4; } // writing and handling error

        close(pipeFD[0][1]); //close writing in process 1
        return 0;
    }

    pid_t pid2 = fork();
```

OS_1212508 > main.c > main(int, char * [])

```c
    pid_t pid2 = fork();
    if (pid2 == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }

    if (pid2 == 0)
    {
        //Child process 2
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[0][1]); //close write of process 1
        close(pipeFD[2][0]); //close read of process 3
        close(pipeFD[2][1]); //close write of process 3
        close(pipeFD[3][0]); //close read of process 4
        close(pipeFD[3][1]); //close write of process 4

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[1][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error

        close(pipeFD[1][1]); //close writing in process 2
        return 0;
    }

    pid_t pid3 = fork();
    if (pid3 == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }

    if (pid3 == 0)
    {
```

OS_1212508 > main.c > main(int, char * [])

```c
    if (pid3 == 0)
    {
        //Child process 3
        close(pipeFD[2][0]); //close read of process 3
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[0][1]); //close write of process 1
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[1][1]); //close write of process 2
        close(pipeFD[3][0]); //close read of process 4
        close(pipeFD[3][1]); //close write of process 4

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[2][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error

        close(pipeFD[2][1]); //close writing in process 3
        return 0;
    }

    pid_t pid4 = fork();
    if (pid4 == -1)   // error handling when fork failure and this return 1
    {
        printf("An error occured with fork!\n");
        return 1;
    }

    if (pid4 == 0)
    {
        //Child process 4
        close(pipeFD[3][0]); //close read of process 4
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[0][1]); //close write of process 1
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[1][1]); //close write of process 2
```

OS_1212508 > C main.c > main(int, char * [])

```c
    if (pid4 == 0)
    {
        //Child process 4
        close(pipeFD[3][0]); //close read of process 4
        close(pipeFD[0][0]); //close read of process 1
        close(pipeFD[0][1]); //close write of process 1
        close(pipeFD[1][0]); //close read of process 2
        close(pipeFD[1][1]); //close write of process 2
        close(pipeFD[2][0]); //close read of process 3
        close(pipeFD[2][1]); //close write of process 3

        childProcess(idMatrix, idXyearMatrix, resultMatrix);

        if(write(pipeFD[3][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error

        close(pipeFD[3][1]); //close writing in process 4
        return 0;
    }

    //Parent process
    //start = clock();

    close(pipeFD[0][1]); //close write of process 1
    close(pipeFD[1][1]); //close write of process 2
    close(pipeFD[2][1]); //close write of process 3
    close(pipeFD[3][1]); //close write of process 4

    gettimeofday(&end, NULL);

    waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
    waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
    waitpid(pid3, NULL, 0); // waiting for process 3 to finish operation
    waitpid(pid4, NULL, 0); // waiting for process 4 to finish operation

```

OS_1212508 > C main.c > main(int, char * [])

```c
    }

    //Parent process
    close(pipeFD[0][1]); //close write of process 1
    close(pipeFD[1][1]); //close write of process 2
    close(pipeFD[2][1]); //close write of process 3
    close(pipeFD[3][1]); //close write of process 4

    gettimeofday(&end, NULL);

    waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
    waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
    waitpid(pid3, NULL, 0); // waiting for process 3 to finish operation
    waitpid(pid4, NULL, 0); // waiting for process 4 to finish operation

    if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1

    if(read(pipeFD[1][0], resultMatrix + N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 2

    if(read(pipeFD[2][0], resultMatrix + 2*N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 3

    if(read(pipeFD[3][0], resultMatrix + 3*N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 4

    close(pipeFD[0][0]); //close read of process 1
    close(pipeFD[1][0]); //close read of process 2
    close(pipeFD[2][0]); //close read of process 3
    close(pipeFD[3][0]); //close read of process 4
    //printMatrix(resultMatrix);

    tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
    printf("\n\nTime is: %fs\n\n", tm);

    return 0;
}

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Time is: 0.004283s

[1] + Done                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-l1bev0an.1en" 1>"/tmp/Microsoft-MIEngine-Out-5yhjhf0o.dtx"
qusay@qusay-VirtualBox:~/CProjects$
```

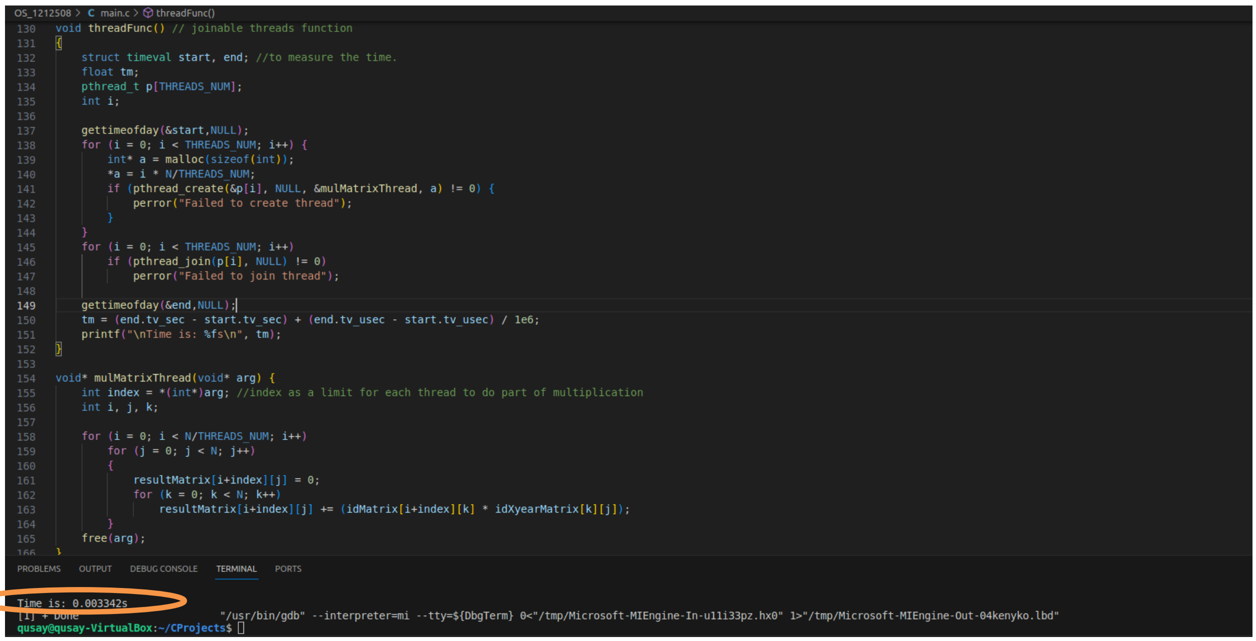
From above snapshots that we used 4 child processes we notice that the time increased to 0.004283s which is 4.283ms instead of decreasing and throughput equals $233.48s^{-1}$ which is less than throughput of 2 child processes.

As a result of trying different number of child processes and based of setting 5G Ram and 4 Cores for my linux VM so the best number of child processes is 2 child process, so incresing number of processes does not make a sense.

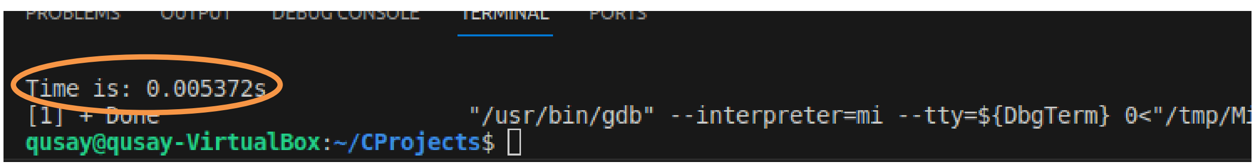
Joinable Threads:

There is below picture of using joinable threads in threadFunc() to do the task, also we implement a new function to do the task "multiplication" that special just for threads its called mulMatrixThread(), then we called thread function after we globalize our matrices.

The time shown in the picture below that for using 2 Threads which is 3.342ms and throughput is 299.22s^{-1}



And the time shown in the picture below that for using 4 Threads which is 5.372ms and throughput is 186.15s^{-1}



As a result of increasing threads the time was increased so that depend on our system and attributes of its components, so using 2 "joinable" threads is better for our case that need 3.342ms and make throughput of 299.22s^{-1}.

* Note: I moved code of multiplying matrices that special for threads to threadFunc and creating threads, joining/detaching in the main function

Detached Threads:

In the pictures bellow we used 2 threads and replacing joining threads with detached and of actually we disable joining to notice the difference in the time.

```
    pthread_t p[THREADS_NUM];                        // array of threads to be
    pthread_attr_t detached;
    pthread_attr_init(&detached);
    pthread_attr_setdetachstate(&detached, PTHREAD_CREATE_DETACHED);
```

```
    //////////////end of initializing/////////
    gettimeofday(&start,NULL);
    //childProcess();

    int m;
    for (m = 0; m < THREADS_NUM; m++) // creating threads loop
    {
        int* a = malloc(sizeof(int));
        *a = m * N/THREADS_NUM;
        if (pthread_create(&p[m], &detached, &threadFunc, a) != 0) {
            perror("Failed to create thread");
        }
        pthread_detach(p[m]); //detaching threads
    }

    /*for (m = 0; m < THREADS_NUM; m++) //joining threads loop
        if (pthread_join(p[m], NULL) != 0)
            perror("Failed to join thread");*/


    pthread_attr_destroy(&detached);
    gettimeofday(&end, NULL);
    tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
    printf("Time from main is: %fs\n", tm);
    //printMatrix(resultMatrix);
    pthread_exit(0);
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Time from main is: 0.002307s
[1] + Done                 "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-lszvasxd.fdg" 1>"/tmp/Microsoft-MIEngine-
qusay@qusay-VirtualBox:~/CProjects$
```

As we see from above the time is 2.307ms and throughput is $433.46s^{-1}$

Now we gonna use 4 threads to show which is better

```
Time from main is: 0.003835s
[1] + Done                 "/usr
qusay@qusay-VirtualBox:~/CProjects$
```

We notice that using 4 threads takes 3.835ms and $260.76s^{-1}$ which are not better than 2 threads.

So in our case the best case is 2 Detached Threads.

* Note: we can’t measure time in detached threads and the reason is main thread terminate its execution before detached and that make complex, but we meagure for all detached and printing the time on the screen as above.

Table of results:

Number/Type	**Native approach**	**Child Processes**	**Joinable Threads**	**Detached Thrads**
Time **Throughput**	5.664ms $176.55s^{-1}$	**2 Processes** 2.361ms $423.55s^{-1}$	**2 Threads** 3.342ms $299.22s^{-1}$	**2 Threads** 2.307ms $433.46s^{-1}$
Time **Throughput**	-	**4 Processes** 4.283ms $233.48s^{-1}$	**4 Threads** 5.372ms $186.15s^{-1}$	**4 Threads** 3.835ms $260.76s^{-1}$

There is in above table we showed the difference between processes and threads in time and throughput.

And we conclude that the best case in child processes is 2 and in both joinable or detached 2 thrads is the best in hand of the fast and throughput, and that is suitable for my linux VM as we mentioned above its properties.

* Note: all code captured in pictures above **have an update** to fits requiered and sent in .rar file that also include this report.