Faculty of Engineering & Technology
Electrical & Computer Engineering Department
ENCS3390 Operating system

Task: Process and Thread Management

Prepared by: Abdalrhman Juber
Number ID: 1211769

Instructor:
Dr. Abdel Salam Sayyad

Section: #1

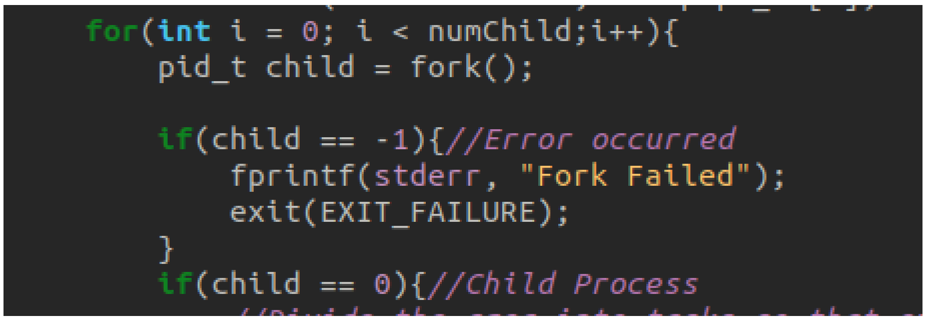
Date: 12/2/2023

Implementation:

In this task we have four parts.I will analysis each part lonely .

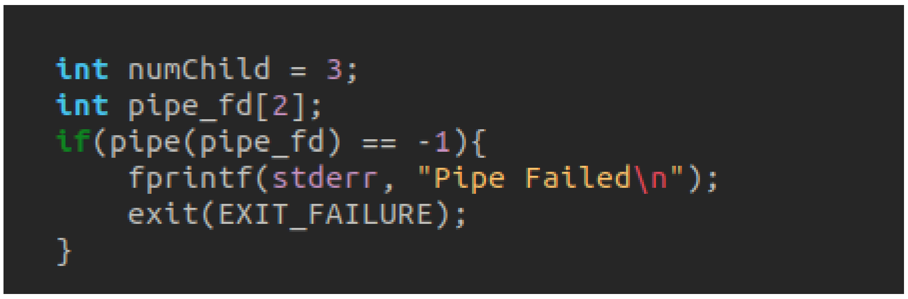
- Part One: **Process Management**

In this part I build a process using fork that will create a child process ,then for communication between the parent and children I use pipe . In the figure 1 and 2 shows the code for implement child and pipe.



```
    for(int i = 0; i < numChild;i++){
        pid_t child = fork();

        if(child == -1){//Error occurred
            fprintf(stderr, "Fork Failed");
            exit(EXIT_FAILURE);
        }
        if(child == 0){//Child Process
```

Figure (1) : shows creating child and make error handling.

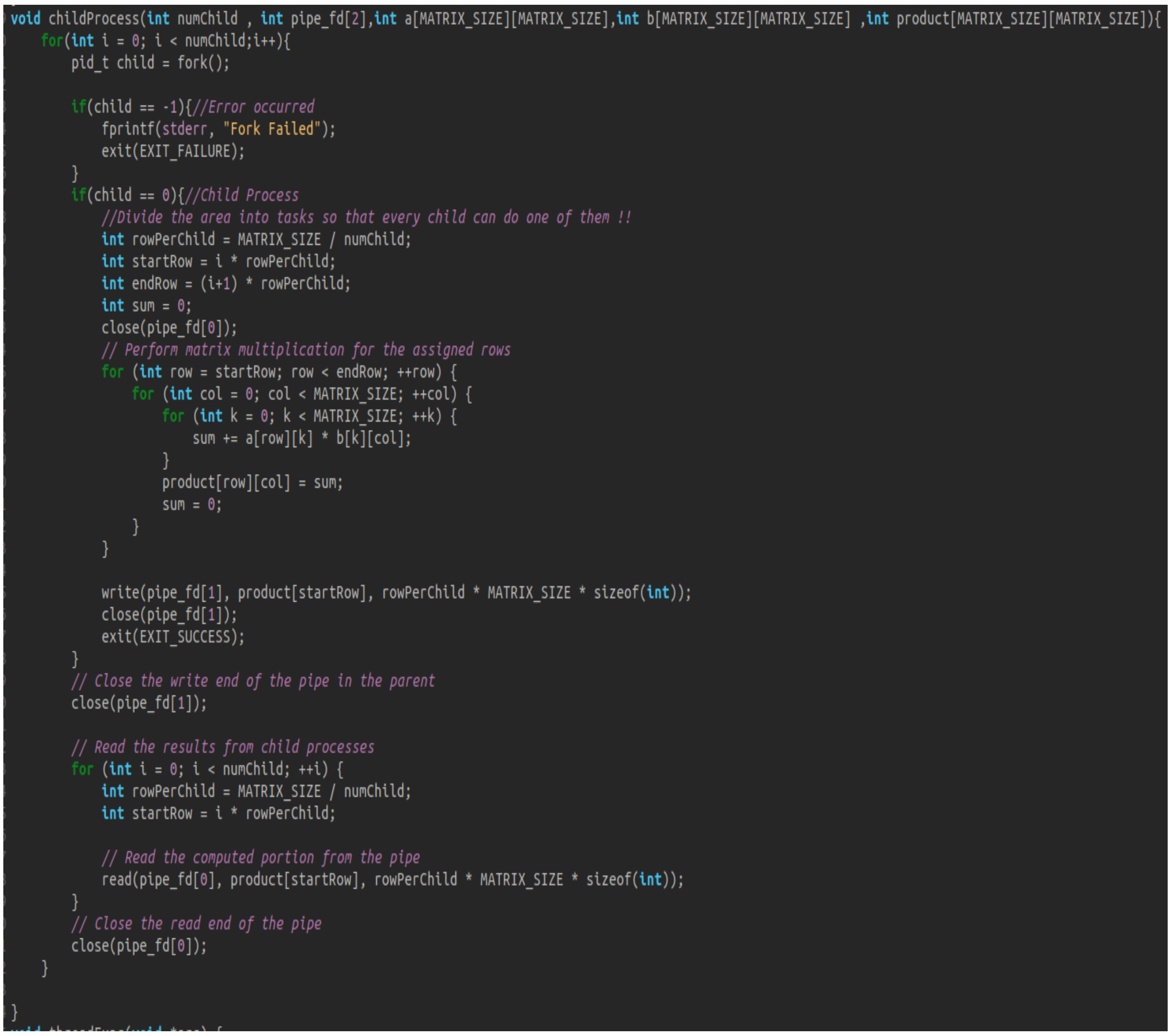


```
int numChild = 3;
int pipe_fd[2];
if(pipe(pipe_fd) == -1){
    fprintf(stderr, "Pipe Failed\n");
    exit(EXIT_FAILURE);
}
```

Figure (2) : shows creating pipe and make error handling.

- **Part One :**

AS requested in the task I made a child-process function.As showing in the figure 3.



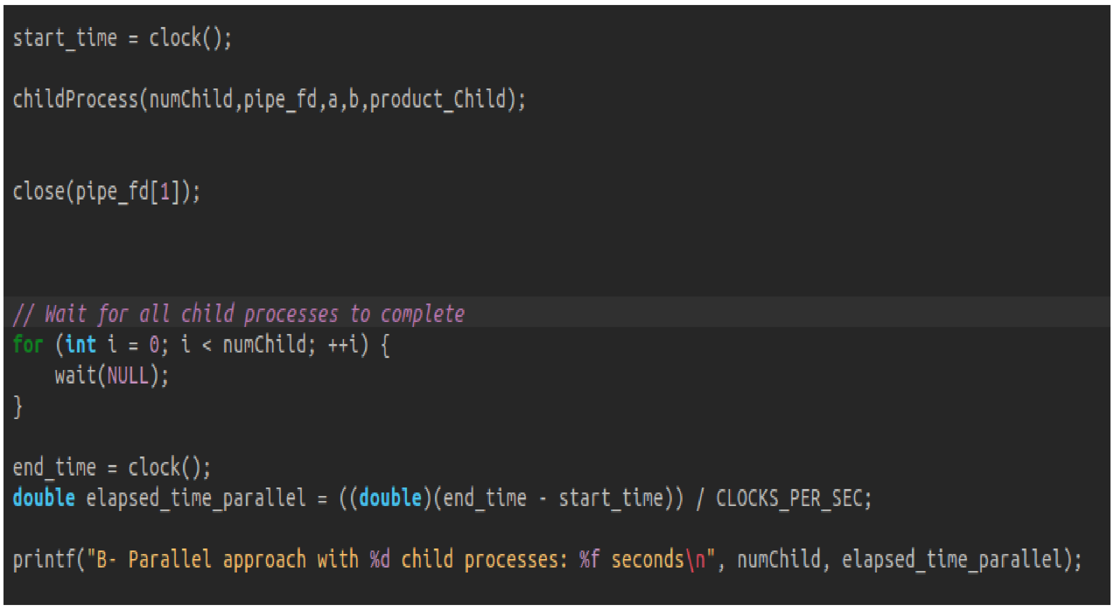
```c
void childProcess(int numChild , int pipe_fd[2],int a[MATRIX_SIZE][MATRIX_SIZE],int b[MATRIX_SIZE][MATRIX_SIZE] ,int product[MATRIX_SIZE][MATRIX_SIZE]){
    for(int i = 0; i < numChild;i++){
        pid_t child = fork();

        if(child == -1){//Error occurred
            fprintf(stderr, "Fork Failed");
            exit(EXIT_FAILURE);
        }
        if(child == 0){//Child Process
            //Divide the area into tasks so that every child can do one of them !!
            int rowPerChild = MATRIX_SIZE / numChild;
            int startRow = i * rowPerChild;
            int endRow = (i+1) * rowPerChild;
            int sum = 0;
            close(pipe_fd[0]);
            // Perform matrix multiplication for the assigned rows
            for (int row = startRow; row < endRow; ++row) {
                for (int col = 0; col < MATRIX_SIZE; ++col) {
                    for (int k = 0; k < MATRIX_SIZE; ++k) {
                        sum += a[row][k] * b[k][col];
                    }
                    product[row][col] = sum;
                    sum = 0;
                }
            }

            write(pipe_fd[1], product[startRow], rowPerChild * MATRIX_SIZE * sizeof(int));
            close(pipe_fd[1]);
            exit(EXIT_SUCCESS);
        }
        // Close the write end of the pipe in the parent
        close(pipe_fd[1]);

        // Read the results from child processes
        for (int i = 0; i < numChild; ++i) {
            int rowPerChild = MATRIX_SIZE / numChild;
            int startRow = i * rowPerChild;

            // Read the computed portion from the pipe
            read(pipe_fd[0], product[startRow], rowPerChild * MATRIX_SIZE * sizeof(int));
        }
        // Close the read end of the pipe
        close(pipe_fd[0]);
    }

}
```

Figure(3) : shown the childProcess function

- Part One:

As show in figure 3 I can change the number of children. When the children ends the parent will wait until all the

Processes end as shown in figure 4.

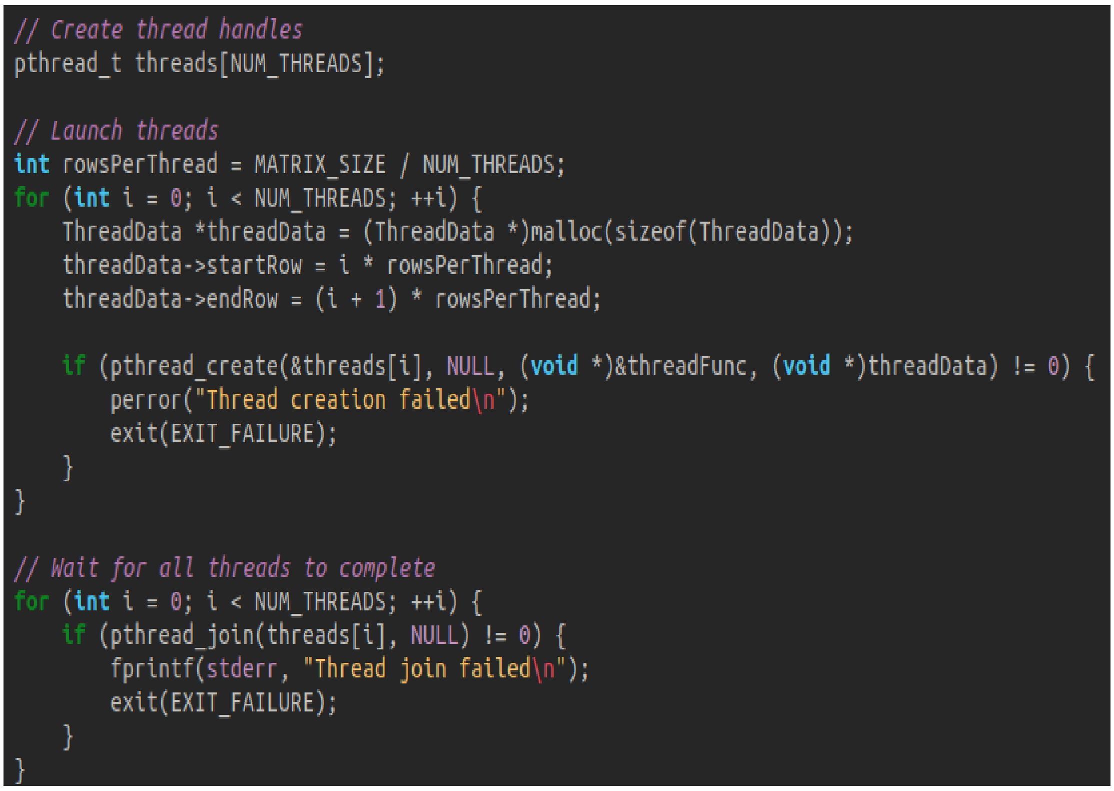


```
start_time = clock();

childProcess(numChild,pipe_fd,a,b,product_Child);


close(pipe_fd[1]);



// Wait for all child processes to complete
for (int i = 0; i < numChild; ++i) {
    wait(NULL);
}

end_time = clock();
double elapsed_time_parallel = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

printf("B- Parallel approach with %d child processes: %f seconds\n", numChild, elapsed_time_parallel);
```

Figure(4) : Parent is waiting for his children

● Part 2: Multithreaded Processing

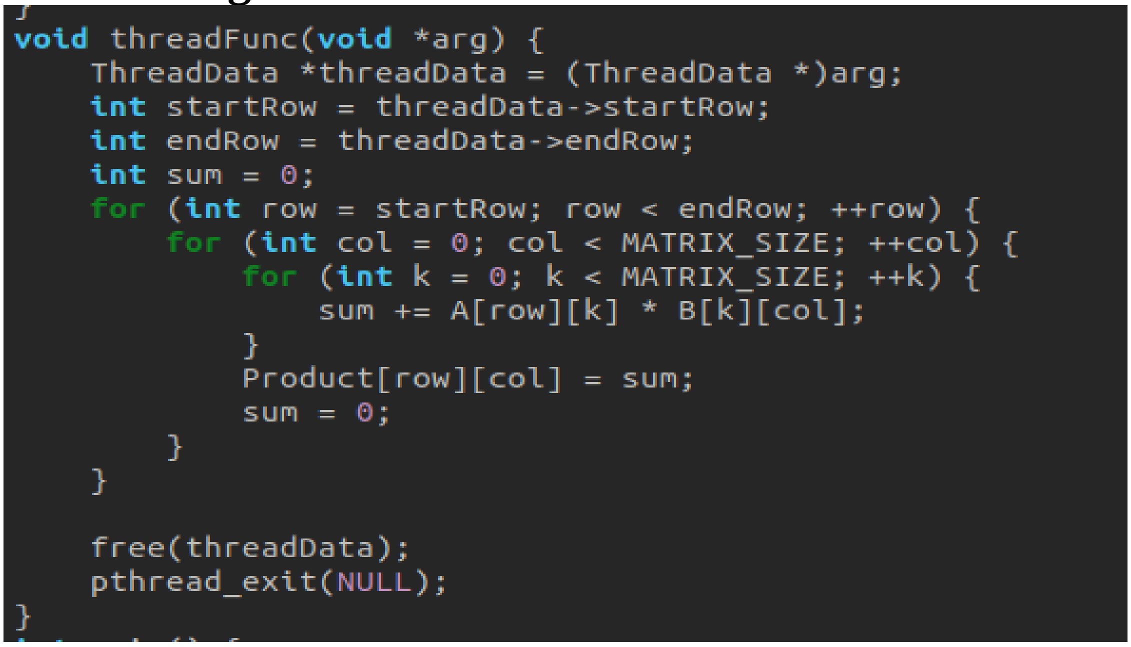
In this part I build multithreded processing that are sharing same data from parent so they are communicate by each other.



```c
// Create thread handles
pthread_t threads[NUM_THREADS];

// Launch threads
int rowsPerThread = MATRIX_SIZE / NUM_THREADS;
for (int i = 0; i < NUM_THREADS; ++i) {
    ThreadData *threadData = (ThreadData *)malloc(sizeof(ThreadData));
    threadData->startRow = i * rowsPerThread;
    threadData->endRow = (i + 1) * rowsPerThread;

    if (pthread_create(&threads[i], NULL, (void *)&threadFunc, (void *)threadData) != 0) {
        perror("Thread creation failed\n");
        exit(EXIT_FAILURE);
    }
}

// Wait for all threads to complete
for (int i = 0; i < NUM_THREADS; ++i) {
    if (pthread_join(threads[i], NULL) != 0) {
        fprintf(stderr, "Thread join failed\n");
        exit(EXIT_FAILURE);
    }
}
```

Figure(5) :Creating threads

As well as, I build a Thread function as required that shown in figure 6.

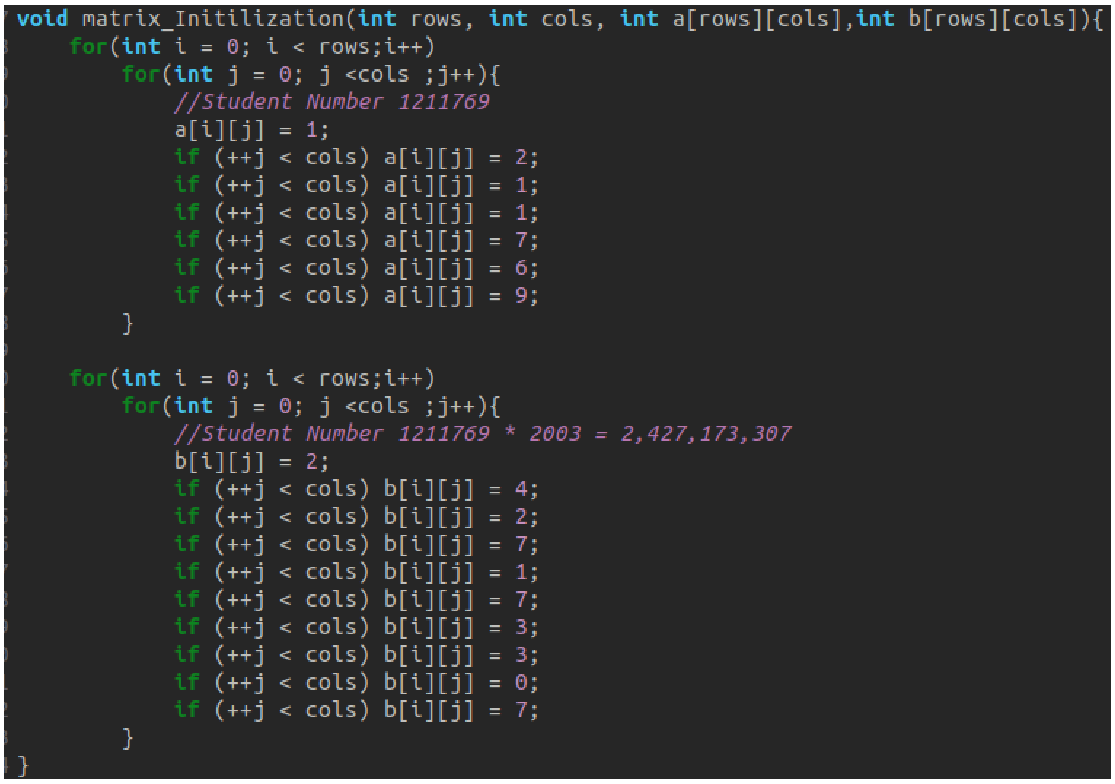


```c
void threadFunc(void *arg) {
    ThreadData *threadData = (ThreadData *)arg;
    int startRow = threadData->startRow;
    int endRow = threadData->endRow;
    int sum = 0;
    for (int row = startRow; row < endRow; ++row) {
        for (int col = 0; col < MATRIX_SIZE; ++col) {
            for (int k = 0; k < MATRIX_SIZE; ++k) {
                sum += A[row][k] * B[k][col];
            }
            Product[row][col] = sum;
            sum = 0;
        }
    }

    free(threadData);
    pthread_exit(NULL);
}
```

Figure(6) : Thread Function

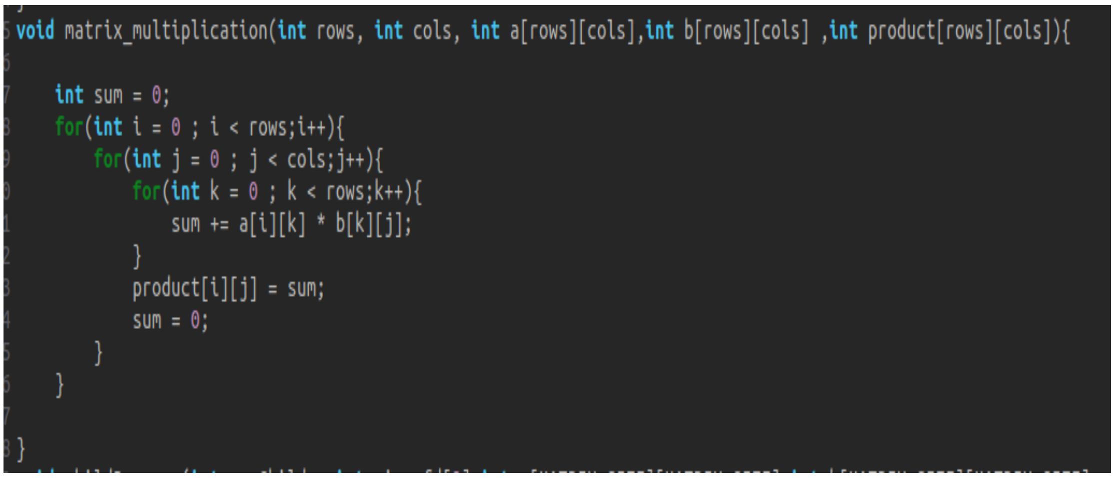
● Part Three: Performance Measurement

For this part I build a matrix with 100*100 as shown in figure 7. And I build matrix multiplication as shown in figure 8.



```
void matrix_Initilization(int rows, int cols, int a[rows][cols],int b[rows][cols]){
    for(int i = 0; i < rows;i++)
        for(int j = 0; j <cols ;j++){
            //Student Number 1211769
            a[i][j] = 1;
            if (++j < cols) a[i][j] = 2;
            if (++j < cols) a[i][j] = 1;
            if (++j < cols) a[i][j] = 1;
            if (++j < cols) a[i][j] = 7;
            if (++j < cols) a[i][j] = 6;
            if (++j < cols) a[i][j] = 9;
        }

    for(int i = 0; i < rows;i++)
        for(int j = 0; j <cols ;j++){
            //Student Number 1211769 * 2003 = 2,427,173,307
            b[i][j] = 2;
            if (++j < cols) b[i][j] = 4;
            if (++j < cols) b[i][j] = 2;
            if (++j < cols) b[i][j] = 7;
            if (++j < cols) b[i][j] = 1;
            if (++j < cols) b[i][j] = 7;
            if (++j < cols) b[i][j] = 3;
            if (++j < cols) b[i][j] = 3;
            if (++j < cols) b[i][j] = 0;
            if (++j < cols) b[i][j] = 7;
        }
}
```

Figure(7) :Creating Matrix with id number until full



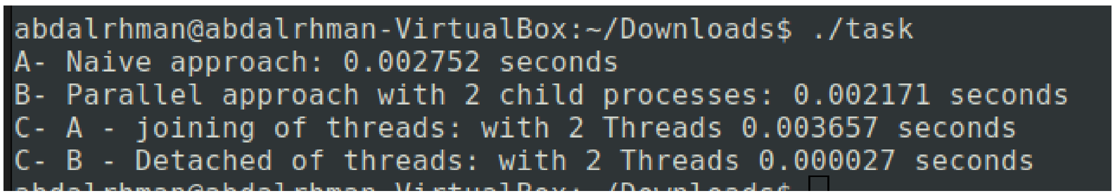
```
void matrix_multiplication(int rows, int cols, int a[rows][cols],int b[rows][cols] ,int product[rows][cols]){

   int sum = 0;
   for(int i = 0 ; i < rows;i++){
       for(int j = 0 ; j < cols;j++){
           for(int k = 0 ; k < rows;k++){
               sum += a[i][k] * b[k][j];
           }
           product[i][j] = sum;
           sum = 0;
       }
   }

}
```

Figure(8) :multiplication matrix

- Measure and compare execution times:

We can see in figure 9 and 10 the time for each child process and thread .

Figure(9) showing child process is faster than threads solution but we know that multiprocessing has higher overhead and lower efficiency than multithreading so the multiprocessing is faster because the thread function has a lot of data to analysis !!.

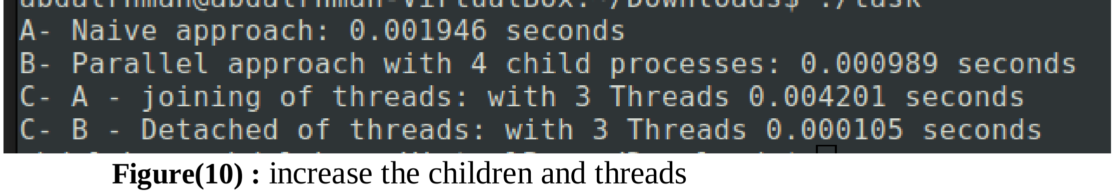


```
abdalrhman@abdalrhman-VirtualBox:~/Downloads$ ./task
A- Naive approach: 0.002752 seconds
B- Parallel approach with 2 child processes: 0.002171 seconds
C- A - joining of threads: with 2 Threads 0.003657 seconds
C- B - Detached of threads: with 2 Threads 0.000027 seconds
```

Figure(9) : Measure and compare

As well we can see in figure 10 that when ever we increase the number of children we in decrease of time and that satisfied with Amdahl’s Law.

But we can see when ever we increase the thread the time increase and that due to the function we created!!

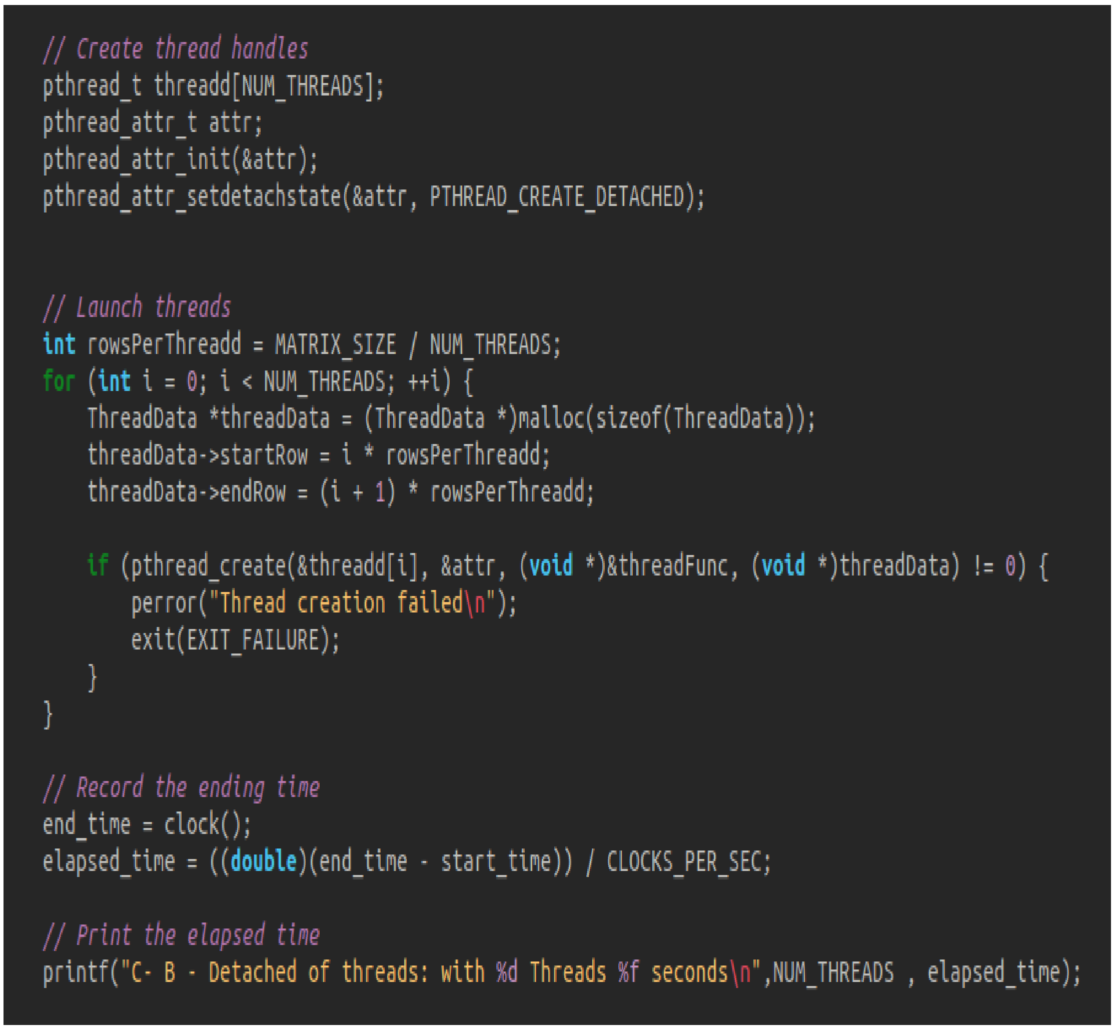


```
abdalrhman@abdalrhman-VirtualBox:~/Downloads$ ./task
A- Naive approach: 0.001946 seconds
B- Parallel approach with 4 child processes: 0.000989 seconds
C- A - joining of threads: with 3 Threads 0.004201 seconds
C- B - Detached of threads: with 3 Threads 0.000105 seconds
```

Figure(10) : increase the children and threads

● Part Four: Thread Management

We create joinable thread as seen in part two .

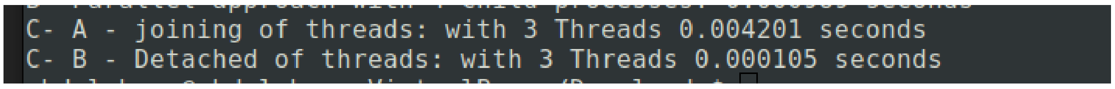


```c
// Create thread handles
pthread_t threadd[NUM_THREADS];
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);


// Launch threads
int rowsPerThreadd = MATRIX_SIZE / NUM_THREADS;
for (int i = 0; i < NUM_THREADS; ++i) {
    ThreadData *threadData = (ThreadData *)malloc(sizeof(ThreadData));
    threadData->startRow = i * rowsPerThreadd;
    threadData->endRow = (i + 1) * rowsPerThreadd;

    if (pthread_create(&threadd[i], &attr, (void *)&threadFunc, (void *)threadData) != 0) {
        perror("Thread creation failed\n");
        exit(EXIT_FAILURE);
    }
}

// Record the ending time
end_time = clock();
elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

// Print the elapsed time
printf("C- B - Detached of threads: with %d Threads %f seconds\n",NUM_THREADS , elapsed_time);
```

Figure(11) : Creating detached thread

The diffrence between joinable and detached threads that the joinable will wait until all thread done. But the detached thread will not wait as its name.

So in our example we can see that the detached thread is faster but it not doing all of matrix to be full !!



```
C- A - joining of threads: with 3 Threads 0.004201 seconds
C- B - Detached of threads: with 3 Threads 0.000105 seconds
```

Figure(12) : Timing of detached and joinig threads

What is the proper/optimal number of child processes or threads?

The number of child processes or threads depends of the application for high level application we can increase the number of processes or threads but in our example (matrix) it does not matter the number.

As well as , In **Amdahl's Law if we keep increasing the number of** child processes or threads it will at the end reach constant time.