



**Faculty of Engineering & Technology  
Electrical & Computer Engineering Department**

**OPERATING SYSTEMS**

**ENCS3390**

**programming task #1**

---

Student Name : Shahd Yahya

Student ID : 1210249

Instructor: Dr. Abdel Salam Sayyad

Section: 1

## This programming task is about matrix multiplication.

- The 2 matrices to be multiplied have the size 100 X 100.
- The first matrix composed of my student number repeated to fill the entire matrix.
- The second matrix is(my student number \* my birth year) repeated to fill the entire matrix.

Here is the implementation of filling the 2 matrices.

```
int main() {
    struct timeval start_t, end_t; // Use struct timeval for time measurements
    int idMatrix[100][100];
    int idxBirthYearMatrix[100][100];
    int multiplicationMatrix[100][100];
    int id = 1210249;
    int ID[10]; // 1D array to contain id numbers
    int size1 = 0; //size of 1D id array
    |
    // fill the id numbers into 1D array
    while(id > 0){
        ID[size1] = id % 10;
        size1++;
        id /= 10;
    }

    // reverse the array because it was filled reversed
    for(int i = 0; i < size1 / 2; i++){
        int temp = ID[i];
        ID[i] = ID[size1 - 1 - i];
        ID[size1 - 1 - i] = temp;
    }

    long long idMulBirth = 1210249L * 2003L;
    int idXbirthYear[20]; // 1D array to contain (id * birth year) numbers
    int size2 = 0; // size of 1D id * BirthYear array

    // fill the id multiplied with the birth year into 1D array
    while(idMulBirth > 0){
        idXbirthYear[size2] = idMulBirth % 10;
        idMulBirth /= 10;
        size2++;
    }
    // reverse the array because it was filled reversed
    for(int i = 0; i < size2 / 2; i++){
        int temp = idXbirthYear[i];
        idXbirthYear[i] = idXbirthYear[size2 - 1 - i];
        idXbirthYear[size2 - 1 - i] = temp;
    }

    // fill the id matrix and id * birth year matrix
    int k = 0, l = 0;
    for(int i = 0; i < 100; i++){
        for(int j = 0; j < 100; j++){
            if(k == size1) k = 0; // when the id ends start over
            if(l == size2) l = 0; // when the id * birth year number ends start over
            idMatrix[i][j] = ID[k];
            idxBirthYearMatrix[i][j] = idXbirthYear[l];
            l++;
            k++;
        }
    }
}
```

## I used 4 approaches of multiplication:

### -The naive approach

multiplying corresponding elements of two matrices and summing the products of the respective rows of the first matrix and columns of the second matrix

-this approach is the slowest approach

Here is the implementation of the naive approach:

```
gettimeofday(&start_t, NULL); // Get the start time

// multiplication process
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        multiplicationMatrix[i][j] = 0;
        for(int k = 0; k < 100; k++){
            multiplicationMatrix[i][j] += (idMatrix[i][k] * idxBirthYearMatrix[k][j]);
        }
    }
}
gettimeofday(&end_t, NULL); // Get the end time
double elapsed_time = (end_t.tv_sec - start_t.tv_sec) + (end_t.tv_usec - start_t.tv_usec) / 1000000.0;
printf("\nThe time of the naive approach is %f ",elapsed_time);
```

I tried calculating the execution time multiple time using gettimeofday() function:

```
The time of the naive approach is 0.002589
Process finished with exit code 0

The time of the naive approach is 0.002525
Process finished with exit code 0

The time of the naive approach is 0.002652
Process finished with exit code 0

The time of the naive approach is 0.002596
Process finished with exit code 0

The time of the naive approach is 0.002655
Process finished with exit code 0
```

The average execution time = 0.0026034 second

Throughput = 1 / average execution time = 384.113

## -Multiprocess approach:

-this approach is better than the naive approach in terms of execution time (because multiple processes are working together in parallel).

Here is the implementation of the multiprocess approach:

```
int rows_per_child = ARRAY_SIZE / NUM_CHILDREN + 1; // Rows assigned to each child(I added 1 to avoid missing any row when the number of children is negative)
int pipe_fds[NUM_CHILDREN][2]; //create a pipe for each child
gettimeofday(&start_t, NULL); // Get the start time

for (int i = 0; i < NUM_CHILDREN; i++) {
    if (pipe(pipe_fds[i]) == -1) {
        perror("Pipe creation failed");
        return 1;
    }
}

pid_t pids[NUM_CHILDREN]; // create pid for each child

for (int i = 0; i < NUM_CHILDREN; i++) {
    pids[i] = fork(); // creating a child
    if (pids[i] < 0) {
        printf("Fork failed");
    }
    else if (pids[i] == 0) {
        // Child process
        close(pipe_fds[i][0]);

        // To store the child result
        int ans_child[rows_per_child][ARRAY_SIZE];

        // Initialize the array elements to zero
        for (int j = 0; j < rows_per_child; j++) {
            for (int k = 0; k < ARRAY_SIZE; k++) {
                ans_child[i][j] = 0;
            }
        }

        // calculate the start and end of the rows assigned to the child
        int start_row = i * rows_per_child;
        int end_row = (i == NUM_CHILDREN - 1) ? ARRAY_SIZE : (i + 1) * rows_per_child;

        // evaluate the child multiplication
        for (int x = start_row; x < end_row; x++) {
            for (int j = 0; j < ARRAY_SIZE; j++) {
                ans_child[x - start_row][j] = 0;
                for (k = 0; k < ARRAY_SIZE; k++) {
                    ans_child[x - start_row][j] += (idMatrix[x][k] * idxBirthYearMatrix[k][j]);
                }
            }
        }

        // Write the result to the pipe
        write(fd: pipe_fds[i][1], buf: ans_child, nbytes: sizeof(ans_child));

        close(pipe_fds[i][1]);
        exit(0); // Exit the child process
    }
}

for (int i = 0; i < NUM_CHILDREN; i++) {
    close(pipe_fds[i][1]);
    waitpid(pids[i], NULL, 0); // wait for child process
    int temp_result[rows_per_child][ARRAY_SIZE]; // To store the child array
    read(pipe_fds[i][0], temp_result, sizeof(temp_result));
    close(pipe_fds[i][0]);
    int start_row = i * rows_per_child;
    int end_row = (i == NUM_CHILDREN - 1) ? ARRAY_SIZE : (i + 1) * rows_per_child;

    // add the partial array to the final array
    for (int x = start_row; x < end_row; x++) {
        for (int j = 0; j < ARRAY_SIZE; j++) {
            multiplicationMatrix[x][j] += temp_result[x - start_row][j];
        }
    }
}

gettimeofday(&end_t, NULL); // Get the end time
double elapsed_time = (end_t.tv_sec - start_t.tv_sec) + (end_t.tv_usec - start_t.tv_usec) / 1000000.0;
int x = NUM_CHILDREN;
printf("\nThe time of the multiprocess approach is %f (%d children) ", elapsed_time, x);
```

I tried calculating the execution time multiple time using gettimeofday() function:

-when the number of processes is 2:

```
The time of the multiprocess approach is 0.001885 (2 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001723 (2 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001798 (2 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001786 (2 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001769 (2 children)
```

```
Process finished with exit code 0
```

The average execution time = 0.0017922 second

Throughput = 1 / average execution time = 557.973

-when the number of processes is 4:

```
The time of the multiprocess approach is 0.001588 (4 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001740 (4 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001679 (4 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001537 (4 children)
```

```
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001648 (4 children)
```

```
Process finished with exit code 0
```

The average execution time = 0.0016384 second

Throughput = 1 / average execution time = 610.352

-when the number of processes is 6:

```
The time of the multiprocess approach is 0.001629 (6 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001544 (6 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001536 (6 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001577 (6 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001654 (6 children)
Process finished with exit code 0
```

The average execution time = 0.001588 second

Throughput = 1 / average execution time = 629.723

-when the number of processes is 8:

```
The time of the multiprocess approach is 0.001780 (8 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001729 (8 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001717 (8 children)
Process finished with exit code 0
```

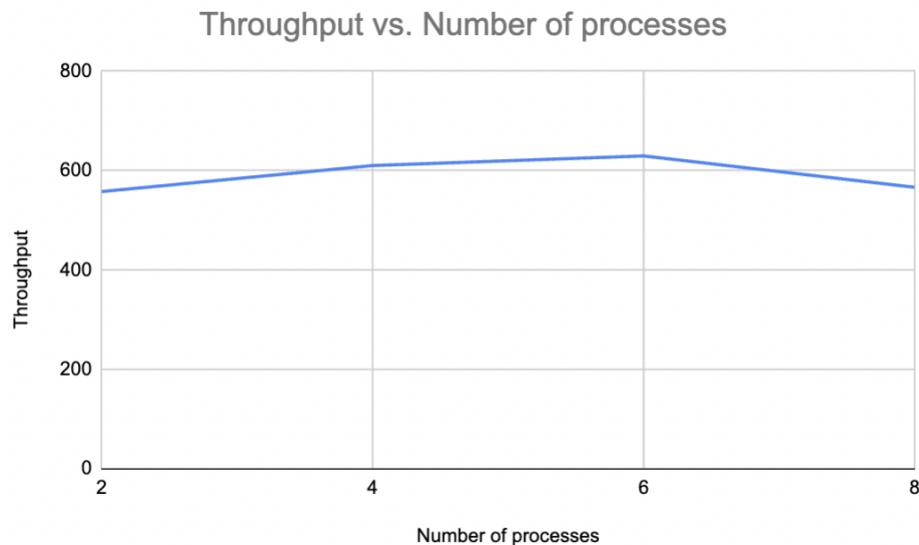
```
The time of the multiprocess approach is 0.001836 (8 children)
Process finished with exit code 0
```

```
The time of the multiprocess approach is 0.001762 (8 children)
Process finished with exit code 0
```

The average execution time = 0.0017648 second

Throughput = 1 / average execution time = 566.636

The best number of processes for the best performance is 6



### -joinable Multithreads approach:

-this approach is the best approach (in terms of execution time and performance)

Here is the implementation of the multithreads approach:

```
int splitIndex = 100 / NumberofThreads;
pthread_t th[NumberofThreads];

gettimeofday(&start_t, NULL); // Get the start time

for (int i = 0; i < NumberofThreads; i++) {
    int start = i * splitIndex;
    int end = (i == NumberofThreads - 1) ? 100 : (i + 1) * splitIndex;
    struct Task *t = (struct Task *)malloc( size: sizeof(struct Task));
    t->start = start;
    t->end = end;
    // Create threads to process tasks
    if (pthread_create(&th[i], NULL, startThread, (void *)t) != 0) {
        printf("Failed to create the thread");
    }
}

// Wait for all threads to finish
for (int i = 0; i < NumberofThreads; i++) {
    if (pthread_join(th[i], NULL) != 0) {
        printf("Failed to join the thread");
    }
}

gettimeofday(&end_t, NULL); // Get the end time

// Calculate the elapsed time in seconds
double elapsed_time = (end_t.tv_sec - start_t.tv_sec) + (end_t.tv_usec - start_t.tv_usec) / 1000000.0;

printf("\nThe time of the multithreading approach is %f seconds", elapsed_time);
```

And here is the startThread function:

```
// struct task to add to the thread pool
typedef struct Task {
    int start, end;
} Task;

// Thread function to process tasks
void *startThread(void *task) {
    struct Task *myTask = (struct Task*)task;

    // Process the task
    for (int x = myTask->start; x < myTask->end; x++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 100; k++) {
                multiplicationMatrix[x][j] += (idMatrix[x][k] * idxBirthYearMatrix[k][j]);
            }
        }
    }

    pthread_exit(NULL);
}
```

I tried calculating the execution time multiple time using gettimeofday() function:

-when the number of threads is 2:

```
The time of the multithreading approach is 0.001145 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.001133 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.001081 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.001091 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.001122 seconds  
Process finished with exit code 0
```

The average execution time = 0.0011144 second

Throughput = 1 / average execution time = 897.343

-when the number of threads is 4:

```
The time of the multithreading approach is 0.000931 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000958 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.001017 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000951 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000816 seconds  
Process finished with exit code 0
```

The average execution time = 0.0009346 second

Throughput = 1 / average execution time = 1069.976

-when the number of threads is 6:

```
The time of the multithreading approach is 0.000770 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000845 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000887 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000779 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000783 seconds  
Process finished with exit code 0
```

The average execution time = 0.0008128 second

Throughput = 1 / average execution time = 1230.315

-when the number of threads is 8:

```
The time of the multithreading approach is 0.000705 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000638 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000773 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000775 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000661 seconds  
Process finished with exit code 0
```

The average execution time = 0.0007104 second

Throughput = 1 / average execution time = 1407.657

-when the number of threads is 10:

```
The time of the multithreading approach is 0.000701 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000697 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000703 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000675 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000691 seconds  
Process finished with exit code 0
```

The average execution time = 0.0006934 second

Throughput = 1 / average execution time = 1442.169

-when the number of threads is 12:

```
The time of the multithreading approach is 0.000654 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000780 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000665 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000680 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000722 seconds  
Process finished with exit code 0
```

The average execution time = 0.0007002 second

Throughput = 1 / average execution time = 1428.163

-when the number of threads is 14:

```
The time of the multithreading approach is 0.000739 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000706 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000752 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000648 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000679 seconds  
Process finished with exit code 0
```

The average execution time = 0.0007048 second

Throughput = 1 / average execution time = 1418.842

-when the number of threads is 16:

```
The time of the multithreading approach is 0.000771 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000778 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000887 seconds  
Process finished with exit code 0
```

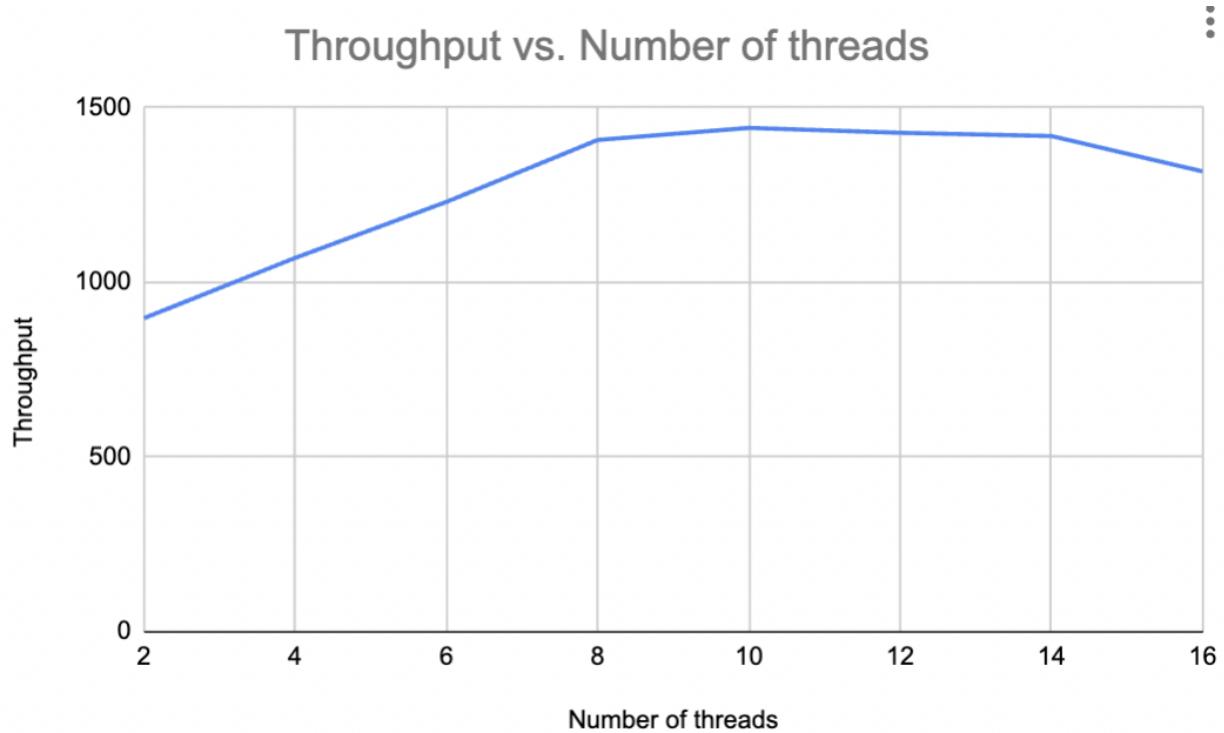
```
The time of the multithreading approach is 0.000740 seconds  
Process finished with exit code 0
```

```
The time of the multithreading approach is 0.000799 seconds  
Process finished with exit code 0
```

The average execution time = 0.000795 second

Throughput = 1 / average execution time = 1317.523

The best number of processes for the best performance is (8-14):



### -detached Multithreads approach:

Detached threads operate independently of the main program. Once created, detached threads do not require explicit waiting for their completion.

Here is the implementation of the multithreads approach:

```
#define NumberOfThreads 4
int multiplicationMatrix[100][100] = {0}; // Result matrix
int idMatrix[100][100];
int idxBirthYearMatrix[100][100];

// struct task to add to the thread pool
typedef struct Task {
    int start, end;
} Task;

void *startThread(void *task) {
    struct Task *myTask = (struct Task*)task;

    // Process the task
    for (int x = myTask->start; x < myTask->end; x++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 100; k++) {
                multiplicationMatrix[x][j] += (idMatrix[x][k] * idxBirthYearMatrix[k][j]);
            }
        }
    }

    pthread_exit(NULL);
}
```

And here is the main program:

```
int splitIndex = 100 / NumberOfThreads;
pthread_t th[NumberOfThreads];

// Loop to create and launch multiple threads
for (int i = 0; i < NumberOfThreads; i++) {
    // Calculate the start and end indices for the current thread
    int start = i * splitIndex;
    int end = (i == NumberOfThreads - 1) ? 100 : (i + 1) * splitIndex;

    // Allocate memory for a task structure to hold thread-specific data
    struct Task *t = (struct Task *)malloc( size: sizeof(struct Task));

    // Set the start and end indices in the task structure
    t->start = start;
    t->end = end;

    // Initialize thread attributes
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    // Set the thread to be detached to operate independently
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    // Create a new thread and pass the startThread function with the task as an argument
    if (pthread_create(&th[i], &attr, startThread, (void *)t) != 0) {
        printf("Failed to create the thread");
    }

    // Destroy thread attributes after thread creation
    pthread_attr_destroy(&attr);
}
```

-In a detached thread approach, measuring execution time can be challenging due to the lack of synchronization with the main program.

-If we do not allow enough time for all threads to execute, the result might be incorrect because some threads may not have finished.

## **Conclusion:**

### Naive Approach:

Method: Multiplying corresponding elements of two matrices without using child processes or threads.

Performance: Slowest approach.

Average Execution Time: 0.0026034 seconds.

This approach is straightforward but lacks parallelism, leading to slower execution.

### Multiple Child Processes:

Method: Using multiple child processes for parallel execution.

Performance: Improved over the naive approach.

Best Performance: 6 processes with an average execution time of 0.001588 seconds.

Increasing the number of processes improved performance, but there may be diminishing returns beyond a certain point due to overhead.

### Multiple Joinable Threads:

Method: Employing multiple joinable threads for parallel execution.

Performance: Best approach among the tested methods.

Best Performance: (8-12) threads with an average execution time of 0.0006934 seconds.

Thread-based parallelism performed better than process-based parallelism in terms of execution time.

### Multiple Detached Threads:

Method: Using multiple detached threads for parallel execution.

Performance: Not explicitly measured due to challenges in timing detached threads.

Detached threads operate independently of the main program, making it challenging to measure execution time accurately.