+

BIRZEIT UNIVERSITY

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Operating Systems ENCS3390

Task 1

Process and Thread Management

Prepared by:

Name: Doaa Hatu **Number:** 1211088

Instructor: Bashar Tahayna

Section: 4

Date: 20/11/2023

Abstract

In this task, we performed a matrix multiplication operation using different approaches (Naïve, processes, joinable threads, and detached threads), then we measured the execution time for each approach, and compared the results to determine which one is better with the least execution time and higher throughput.

Table of Contents

Table of figures

List of tables

Part 1: Naïve approach

In this approach, I wrote a simple c code to calculate the result of two matrix multiplication according to an algorithm, and using for loops for rows and columns.

I calculated its time in the main using the getCurrentTime() system call in the main function.

This approach tends to have the highest execution time - O(n3) -

```c
// A Function for matrix multiplication - Naive approach
void MatrixMultiplication(int m1[MATRIXSIZE][MATRIXSIZE], int m2[MATRIXSIZE][MATRIXSIZE], int result[MATRIXSIZE][MATRIXSIZE])
{

    for(int i=0 ; i<MATRIXSIZE ; i++)
    {
        for(int j=0 ; j<MATRIXSIZE ; j++)
        {
            result[i][j]=0;
            for(int k=0 ; k<MATRIXSIZE ; k++)
            {
                result[i][j] += m1[i][k]*m2[k][j];

            }
        }

    }
}
```

```c
// Naive approach
start_time = getCurrentTime();
MatrixMultiplication(M1, M2, result);
end_time = getCurrentTime();
printf("Naive Approach: Execution Time = %lld microseconds\n", end_time - start_time);
// Naive approach
double throughput1 = total_elements / (end_time - start_time);
printf("Naive Approach: Throughput = %.2f elements/microsecond\n\n", throughput1);
```

Figure 1: Naïve approach code.

Part 2: Process-based solution

For the process-based solution, I used a multiplication function similar to the one in the Naive approach, but depending on the first and last row of the matrix.

In the main, I used pipes for communication between the child process and the parent process.

The time is calculated using getCurrentTime() system call. I calculated the time and the throughput according to specific formulas, depending on the start and end time, and the total elements of matrices.

```
// A Function for process-based matrix multiplication
void processMatrixMultiplication(int m1[MATRIXSIZE][MATRIXSIZE], int m2[MATRIXSIZE][MATRIXSIZE], int result[MATRIXSIZE][MATRIXSIZE], int startRow, int endRow, int fd[2])
{

    for(int i=startRow ; i<endRow ; i++)
    {
        for(int j=0 ; j<MATRIXSIZE ; j++)
        {
            result[i][j]=0;
            for(int k=0 ; k<MATRIXSIZE ; k++)
            {
                result[i][j] += m1[i][k]*m2[k][j];

            }
        }
    }

}
```

```
// Processes approach
start_time = getCurrentTime();
// Create a pipe
int fd[2];
// fd[0] for read
// fd[1] for write
if (pipe(fd) == -1)
{
    perror("pipe");
    exit(EXIT_FAILURE);
}

// Dividing the matrix into parts, each process will have specific number of rows
int rowsPerProcess = MATRIXSIZE/PROCESSESNUM;
int remainingRows = MATRIXSIZE%PROCESSESNUM;
int startRow = 0 ;

for (int i = 0; i < PROCESSESNUM; ++i)
{

    int endRow = startRow + rowsPerProcess + (i < remainingRows ? 1 : 0);

    // Fork a child process
    pid_t child_pid = fork();

    // Check the return value of pipe
    if (pipe(fd) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    if (child_pid == -1)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }
```

```
    if (child_pid == 0)
    {
        // Child process
        processMatrixMultiplication(M1,M2,result, startRow, endRow,fd);
        exit(0);

    }


    else
    {
        // Parent process
        close(fd[1]); // Close the write end of the pipe in the parent process
        // Read data from the pipe
        int Result[MATRIXSIZE][MATRIXSIZE];
        read(fd[0], Result, sizeof(Result));
        // Close the read end of the pipe in the parent process
        close(fd[0]);
        // Wait for the child process to finish
        waitpid(child_pid, NULL, 0);

        // Child process
        close(fd[0]); // Close the read end of the pipe in the child process
        // Write data into the pipe
        write(fd[1], result, sizeof(result));
        // Close the write end of the pipe in the child process
        close(fd[1]);

    }

    startRow = endRow;
}

// Wait for processes to finish
for (int i = 0; i < PROCESSESNUM; i++)
{
    wait(NULL);
}
end_time = getCurrentTime();
printf("Processes Approach: Execution Time = %lld microseconds\n", end_time - start_time);
```

Figure 2: Process-based approach code.

Part 3: Threaded-based solution – Joinable threads

For the joinable threaded-based solution, I implemented a function to perform matrix multiplication, here, each thread will take a specific number of rows to work on it.

In the main, I created the pthreads, depending on their IDs, then measured the time using getCurrentTime().

```
// A Function for thread-based matrix multiplication
void* threadMatrixMultiplication(void* arg)
{
    // Define the thread ID
    int threadID = *((int*)arg);

    // Calculate the number of rows per Thread
    int rowsPerThread = MATRIXSIZE / THREADSNUM;

    int rstart = threadID * rowsPerThread;
    int rend = (threadID == THREADSNUM - 1) ? MATRIXSIZE : (threadID + 1) * rowsPerThread;

    // Matrix multiplication for rows
    int localResult[MATRIXSIZE][MATRIXSIZE];
    for (int i = rstart; i < rend; i++)
    {
        for (int j = 0; j < MATRIXSIZE; j++)
        {
            localResult[i][j] = 0;
            for (int k = 0; k < MATRIXSIZE; k++)
            {
                localResult[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }

    // Copy local result to the global result matrix
    for (int i = rstart; i < rend; i++)
    {
        for (int j = 0; j < MATRIXSIZE; j++)
        {
            result[i][j] = localResult[i][j];
        }
    }

    pthread_exit(NULL);
}
```

```
// Pthreads - Joinable
start_time = getCurrentTime();

pthread_t threads[THREADSNUM];
int threadID[THREADSNUM]; // Array to store thread IDs

// Create threads
for (int i = 0; i < THREADSNUM; i++)
{
    threadID[i] = i;
    pthread_create(&threads[i], NULL, threadMatrixMultiplication, &threadID[i]);
}

// Wait for threads to finish
for (int i = 0; i < THREADSNUM; i++)
{
    pthread_join(threads[i], NULL);
}

end_time = getCurrentTime();
printf("Pthreads - Joinable Approach: Execution Time = %lld microseconds\n", end_time - start_time);
// Pthreads - Joinable
double throughput3 = total_elements / (end_time - start_time);
printf("Pthreads - Joinable Approach: Throughput = %.2f elements/microsecond\n\n", throughput3);
```

Figure 3: Joinable threads-based approach code.

Part 4: Threaded-based solution – Detached threads

For the detached threaded-based solution, it is a bit different, measuring time for detached threads is challenging, since they operate independently, and the main thread does not wait for them to finish.

The pthread_join function which is used for measuring time (I used it for joinable threads) cannot be used here.

I used the same matrix multiplication function I have used for joinable threads. In the main, I created the detached threads with their attributes, and started measuring time using the getCurrentTime() system call. Here, we actually measure the time of creating these threads, because we cannot know their exact execution time.

```
// Pthreads - Detached
start_time = getCurrentTime();

pthread_t Dthreads[THREADSNUM];
int* DthreadID[THREADSNUM]; // Array to store thread IDs
pthread_attr_t attributes;

// Set thread attributes
pthread_attr_init(&attributes);
pthread_attr_setdetachstate(&attributes, PTHREAD_CREATE_DETACHED);


// Create threads

for(int i=0 ; i<THREADSNUM ; i++)
{
    DthreadID[i] = malloc(sizeof(int));
    *DthreadID[i] = i;
    if(pthread_create(&Dthreads[i], &attributes, threadMatrixMultiplication, (void*)DthreadID[i]) != 0)
    {
        fprintf(stderr, "Failed to create threads.\n");
        exit(EXIT_FAILURE);

    }

}

// Destroy the thread attributes after thread creation
pthread_attr_destroy(&attributes);

end_time = getCurrentTime();
printf("Pthreads - Detached Approach: Execution Time = %lld microseconds\n", end_time - start_time);
// Pthreads - Detached
double throughput4 = total_elements / (end_time - start_time);
printf("Pthreads - Detached Approach: Throughput = %.2f elements/microsecond\n", throughput4);
```

Figure 4: Detached threads-based approach code.

Results Analysis and Comparison

Execution time:

Approach	Execution time
Naïve approach	14876 microseconds
Process-based solution – 4 processes	10335 microseconds
Process-based solution – 3 processes	10855 microseconds
Process-based solution – 5 processes	15481 microseconds
Joinable Threaded-based – 4 threads	6691 microseconds
Joinable Threaded-based – 3 threads	10351 microseconds
Joinable Threaded-based – 5 threads	4926 microseconds
Joinable Threaded-based – 6 threads	7310 microseconds
Detached Threaded-based – 4 threads	94 microseconds
Detached Threaded-based – 3 threads	117 microseconds
Detached Threaded-based – 5 threads	178 microseconds

Table 1: Execution time comparison.

According to the results in the above table, we can see that the Naïve approach has the highest execution time -not practical-.

For the process-based solution, it comes in the second place. We can see that the optimal number of child processes is 4, since this number gives the minimum execution time in comparison with 5 processes and 3 processes. Incrementing the number over 4, and decrementing it, will lead to a higher execution time.

For the joinable threads, it seems that 5 is the optimal number for threads, since it gives the least execution time, incrementing or decrementing the number of these threads will give a higher execution time.

For the detached threads, they seem to give the least execution time of 94 microseconds using 4 threads, if we try to increment this number to 5 or more, and decrement it to 3 or less, it will give a higher execution time.

Throughput:

Approach	Throughput
Naïve approach	67.22 elements / microseconds
Process-based solution – 4 processes	96.76 elements / microseconds
Process-based solution – 3 processes	92.12 elements / microseconds
Process-based solution – 5 processes	64.60 elements / microseconds
Joinable Threaded-based – 4 threads	149.45 elements / microseconds
Joinable Threaded-based – 3 threads	96.61 elements / microseconds
Joinable Threaded-based – 5 threads	203 elements / microseconds
Joinable Threaded-based – 6 threads	136.80 elements / microseconds
Detached Threaded-based – 4 threads	10638.30 elements / microseconds
Detached Threaded-based – 3 threads	8547.01 elements / microseconds
Detached Threaded-based – 5 threads	5617.98 elements / microseconds

Table 2: Throughput comparison.

From the throughput table above, we can conclude to similar results as the previous table. The least execution time means higher throughput, and vise versa.

The Naïve approach almost gives the least throughput, and the detached threads with 4 number of threads gives the higher throughput.

For processes, as I said before, the best number of processes to give a higher throughput is 4.

For joinable threads, it was different from joinable ones, since they give a better performance with 5 threads, and a higher throughput.