BIRZEIT UNIVERSITY

OPERATING SYSTEMS

ENCS3390

The First Programming Task

Prepared by:

Abdelrhman Abed 1193191

Instructor: **Dr . Abdel Salam Sayyad**

Section: 2

THE CODE:

```c
//Abdelrhman Abed 1193191
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>
#define MS 100
typedef struct {
    int start_row;
    int end_row;
    int matrix_a[MS][MS];
    int matrix_b[MS][MS];
    int result[MS][MS];
} thread_data;

void* worker_thread(void* arg) {
    thread_data* data = (thread_data*)arg;

    for (int i = data->start_row; i < data->end_row; ++i) {
        for (int j = 0; j < MS; ++j) {
            data->result[i][j] = 0;
            for (int k = 0; k < MS; ++k) {
                data->result[i][j] += data->matrix_a[i][k] * data->matrix_b[k][j];
            }
        }
    }

    return 0;
```

1- <stdio.h> and <stdlib.h> for standard input/output and general utilities.

2 - <sys/time.h> for time-related functions.

3 - <sys/wait.h> for process-related functions.

4 - <unistd.h> for POSIX operating system API.

5 - <pthread.h> for multithreading support.

6 - #define MS 100: A macro definition. MS is defined as 100, and it is used to specify the size of the matrices.

“thread_data “: Every thread needs its own set of information, which is specified by this structure. It consists of:

1 - start_row and end_row for specifying the range of rows the thread will work on.

2- matrix_a and matrix_b for input matrices.

3- result for the resulting matrix of the multiplication.

A structure called "thread_data" that holds data required by every thread is described. Matrix_a and matrix_b are the input matrices; start_row and end_row define the range of rows the thread will work on; and a result matrix stores the outcome of the multiplication.

The function "worker_thread" is introduced once the paragraph comes to an end. Every thread carries out this method, which accepts a void pointer as an input and casts it to a thread_data pointer. The worker_thread function stores the outcome in the result matrix after completing matrix multiplication for the designated rows in matrix_a and matrix_b.

```c
void parallel_threaded(int thread_count, int matrix_a[MS][MS], int matrix_b[MS][MS], int result[MS][MS]) {
    pthread_t treadhandles[thread_count];
    thread_data thread_data[thread_count];

    for (int i = 0; i < thread_count; ++i) {
        thread_data[i].start_row = i * (MS / thread_count);
        thread_data[i].end_row = (i + 1) * (MS / thread_count);
        for (int j = 0; j < MS; ++j) {
            for (int k = 0; k < MS; ++k) {
                thread_data[i].matrix_a[j][k] = matrix_a[j][k];
                thread_data[i].matrix_b[j][k] = matrix_b[j][k];
            }
        }
    }

    for (int i = 0; i < thread_count; ++i) {
        if (pthread_create(&treadhandles[i], NULL, worker_thread, &thread_data[i]) != 0) {
            fprintf(stderr, "create failed\n");
            exit(0);
        }
    }

    for (int i = 0; i < thread_count; ++i) {
        if (pthread_join(treadhandles[i], NULL) != 0) {
            fprintf(stderr, "join failed\n");
            exit(0);
        }
    }

```


```c
        if (pthread_join(treadhandles[i], NULL) != 0) {
            fprintf(stderr, "join failed\n");
            exit(0);
        }
    }
```


```c

    for (int i = 0; i < thread_count; ++i) {
        for (int j = thread_data[i].start_row; j < thread_data[i].end_row; ++j) {
            for (int k = 0; k < MS; ++k) {
                result[j][k] = thread_data[i].result[j][k];
            }
        }
    }
}
```

The “parallel_threaded “function you provided seems to be the main function for performing parallel matrix multiplication using threads.

This part declares an array of thread handles “treadhandles” and an array of “thread_data” structures “thread_data”. each thread will have its own data structure to work on a specific range of rows.

the “thread_data” structures. It divides the rows of the matrices evenly among the threads. The nested loops copy the values from the input matrices “matrix_a and matrix_b” to the corresponding matrices in the “thread_data “ structure.

This loop creates threads using “pthread_create” each thread is assigned a specific range of rows to work on, and it will execute the “worker_thread “function.

after creating the threads, this loop waits for each thread to finish using “pthread_join” this ensures that the main thread doesn't proceed until all worker threads have completed their tasks.

this loop copies the results from the “thread_data” structures back to the result matrix.This function assumes that the matrices matrix_a, matrix_b, and result are properly initialized and have the correct dimensions.Be cautious about potential race conditions or other concurrency issues that might arise in a multithreaded environment. Ensure that the shared data is properly protected if necessary.This code provides a basic framework for parallel matrix multiplication, but depending on the context, you might need to further optimize and handle edge cases.

```c
void parallel_detache(int thread_count, int matrix_a[MS][MS], int matrix_b[MS][MS], int result[MS][MS]) {
    pthread_t treadhandles[thread_count];
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    thread_data thread_data[thread_count];

    for (int i = 0; i < thread_count; ++i) {
        thread_data[i].start_row = i * (MS / thread_count);
        thread_data[i].end_row = (i + 1) * (MS / thread_count);
        for (int j = 0; j < MS; ++j) {
            for (int k = 0; k < MS; ++k) {
                thread_data[i].matrix_a[j][k] = matrix_a[j][k];
                thread_data[i].matrix_b[j][k] = matrix_b[j][k];
            }
        }
    }
    for (int i = 0; i < thread_count; ++i) {
        if (pthread_create(&treadhandles[i], &attr, worker_thread, &thread_data[i]) != 0) {
            fprintf(stderr, "create failed\n");
            exit(0);
        }
    }

    usleep(10000);

    for (int i = 0; i < thread_count; ++i) {
```

```c
    usleep(10000);

    for (int i = 0; i < thread_count; ++i) {
        for (int j = thread_data[i].start_row; j < thread_data[i].end_row; ++j) {
            for (int k = 0; k < MS; ++k) {
                result[j][k] = thread_data[i].result[j][k];
            }
        }
    }
}
```

Although the "parallel_detach" function you've supplied and the earlier "parallel_threaded" function have certain similarities, the usage of detached threads causes some significant changes.

This section initializes a thread attribute (attr), an array of "thread_data" structures (thread_data), and an array of thread handles (treadhandles). Disconnected Thread Setup: "pthread_attr_setdetachstate" is used to set the detached state of the threads. The resources of detached threads are automatically relinquished once they are finished, and they do not require explicit joining. the "thread_data" structures in a manner akin to that of the earlier function. Using "pthread_create" and the detached property, this loop generates threads. Threads won't be explicitly connected; instead, they will operate independently..

This uses "usleep" to induce a 10,000 microsecond (10 millisecond) delay. This was probably implemented so that the detached threads may finish their calculations before the main thread moved on.This loop replicates the results from the thread_data structures back to the result matrix, much like the preceding function did.

Using detachable threads makes thread management easier since the main thread does not have to wait for each thread to finish. Despite adding a purposeful delay, the "usleep" function is often not a dependable method of maintaining thread synchronization. It would be better to use suitable synchronization mechanisms if needed. Detachable threads can be useful in some circumstances, despite their own set of concerns. Ensure that they won't access any data that could be deallocated before detached threads are finished.

```c
void parallel_m(int process_count, int matrix_a[MS][MS], int matrix_b[MS][MS], int result[MS][MS])
    pid_t process_ids[process_count];

    for (int i = 0; i < process_count; ++i) {
        int start_row = i * (MS / process_count);
        int end_row = (i + 1) * (MS / process_count);

        pid_t child_pid = fork();

        if (child_pid == -1) {
            fprintf(stderr, "fork failed\n");
            exit(0);
        }

        if (child_pid == 0) {
            for (int row = start_row; row < end_row; ++row) {
                for (int col = 0; col < MS; ++col) {
                    result[row][col] = 0;
                    for (int k = 0; k < MS; ++k) {
                        result[row][col] += matrix_a[row][k] * matrix_b[k][col];
                    }
                }
            }
            exit(1);
        } else {
            process_ids[i] = child_pid;
        }
    }

```

```c
            }
        }

        for (int i = 0; i < process_count; ++i) {
            int status;
            waitpid(process_ids[i], &status, 0);
        }
}

```

The "parallel_m" function you provide creates several child processes to do matrix multiplication using the "fork" system call.

To keep track of the children processes, the function first declares an array of process IDs, called "process_ids."

As the loop iterates, a child process forks. The child process then performs matrix multiplication within the designated row range. The program ends and an error is reported if "fork" returns -1. The child process does matrix multiplication on a subset of rows indicated by "start_row" and "end_row" inside the block when "child_pid == 0". With "exit(1)," the child process ends after finishing its task. The parent process maintains the child process ID in the “process_ids” array in the otherwise block “child_pid!= 0“. Once all child processes have been forked, the parent process uses “waitpid” to wait for each child process to complete. Every child process is in responsible of a certain range of rows in the result matrix and runs on its own.

The parent process waits for each child process to finish before continuing. Synchronization is preserved in this way, and the parent process is prevented from continuing while the child processes are still active. Carefully weigh the potential overhead of establishing several procedures. Compared to producing new threads, starting a new process requires replicating the complete process, which might need more resources. Similar to multithreading, if processes share resources or data structures, proper synchronization strategies would be needed.

```
int main()  {

    int matrix_a[MS][MS];
    int matrix_b[MS][MS];
    int result[MS][MS];

    int student_number [7] = {1,1,9,3,1,9,1};
    int student_numberandbirthyear [10]= {2,3,8,7,5,7,5,1,9,1};
    int con1=0;
    int con2=0;
    for(int i=0; i<MS ; i++){
        for(int j=0; j<MS; j++){
            matrix_a[i][j]=student_number[con1];
            con1++;
            if(con1 == 7){
                con1=0;
            }
        }
    }
    for(int i=0; i<MS ; i++){
        for(int j=0; j<MS; j++){
            matrix_b[i][j]=student_numberandbirthyear[con2];
            con2++;
            if(con2 == 10){
                con2=0;
            }
        }
    }

```

```
    }

    struct timeval start, end;

    gettimeofday(&start, NULL);
```

Two matrices, "matrix_a and matrix_b," with dimensions of MS x MS, are initialized in this section of the code. The "student_number" and "student_numberandbirthyear" arrays are cycled through in order to determine the values in these matrices. The rows are represented by the outer loop, while the columns are represented by the inner loop.

Time Measurement: Prior to executing any matrix operations, the beginning time (start) is recorded using the "gettimeofday" function.

The aforementioned code section initializes the time measurement and sets up the matrices for a subsequent operation.

```c
     for (int i = 0; i < MS; ++i) {     //  naive matrix multiplication
        for (int j = 0; j < MS; ++j) {
            result[i][j] = 0;
            for (int k = 0; k < MS; ++k) {
                result[i][j] += matrix_a[i][k] * matrix_b[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);
    double timedifference = (((double)(end.tv_sec - start.tv_sec)) + ((double)(end.tv_usec - start.tv_usec))) / 1000000.0;
    printf("the naive approach time: %lf seconds\n", timedifference);

    for (int thread_count = 2; thread_count <= 4; thread_count *= 2) {
        gettimeofday(&start, NULL);
        parallel_threaded(thread_count, matrix_a, matrix_b, result);
        gettimeofday(&end, NULL);
        double timedifferencethread = (((double)(end.tv_sec - start.tv_sec)) + ((double)(end.tv_usec - start.tv_usec))) / 1000000.0;
        printf("parallel (threads=%d) time: %lf seconds\n", thread_count, timedifferencethread);
    }
    for (int process_count = 2; process_count <= 4; process_count *= 2) {
        gettimeofday(&start, NULL);
        parallel_m(process_count, matrix_a, matrix_b, result);
        gettimeofday(&end, NULL);
        double timedifferenceproc = (((double)(end.tv_sec - start.tv_sec)) + ((double)(end.tv_usec - start.tv_usec))) / 1000000.0;
        printf("parallel (processes=%d) time: %lf seconds\n", process_count, timedifferenceproc);
}
for (int thread_count = 2; thread_count <= 4; thread_count *= 2) {
                    gettimeofday(&start, NULL);
```

```c
for (int thread_count = 2; thread_count <= 4; thread_count *= 2) {
                    gettimeofday(&start, NULL);
                    parallel_detache(thread_count, matrix_a, matrix_b, result);
                    gettimeofday(&end, NULL);
                    double timedifferencedeta = (((double)(end.tv_sec - start.tv_sec)) + ((double)(end.tv_usec - start.tv_usec))) / 1000000.0;
                    printf("parallel (detached threads=%d) time: %lf seconds\n", thread_count, timedifferencedeta);
                }
    return 0;
}
```

This part of the code measures the time required by each method for doing matrix multiplication utilizing naïve, threaded, parallel processes, and detached threads.

Multiplication of Naive Matrix: Three stacked loops and a Naive method of matrix multiplication are used in this code block. It calculates and outputs the needed amount of time.

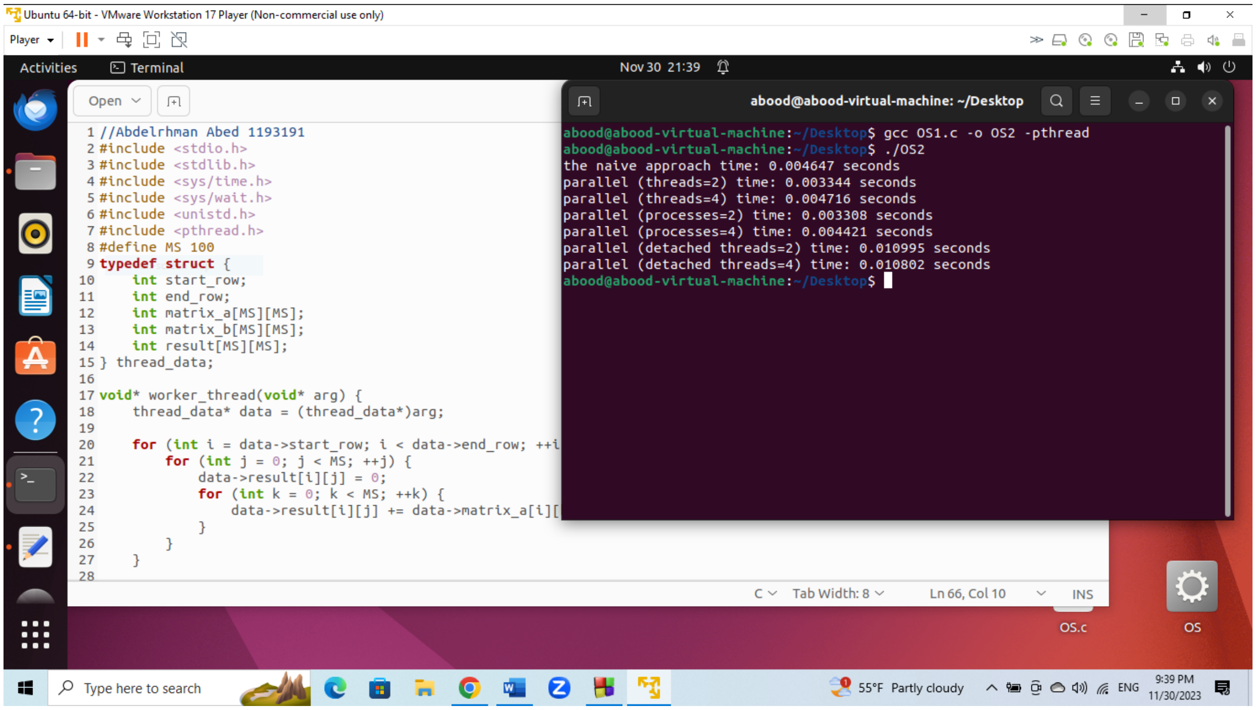
Threaded Matrix Multiplication: This loop uses a variable number of threads (2, 4) to conduct matrix multiplication using the "parallel_threaded" function. It prints the findings and calculates the time needed for each instance.

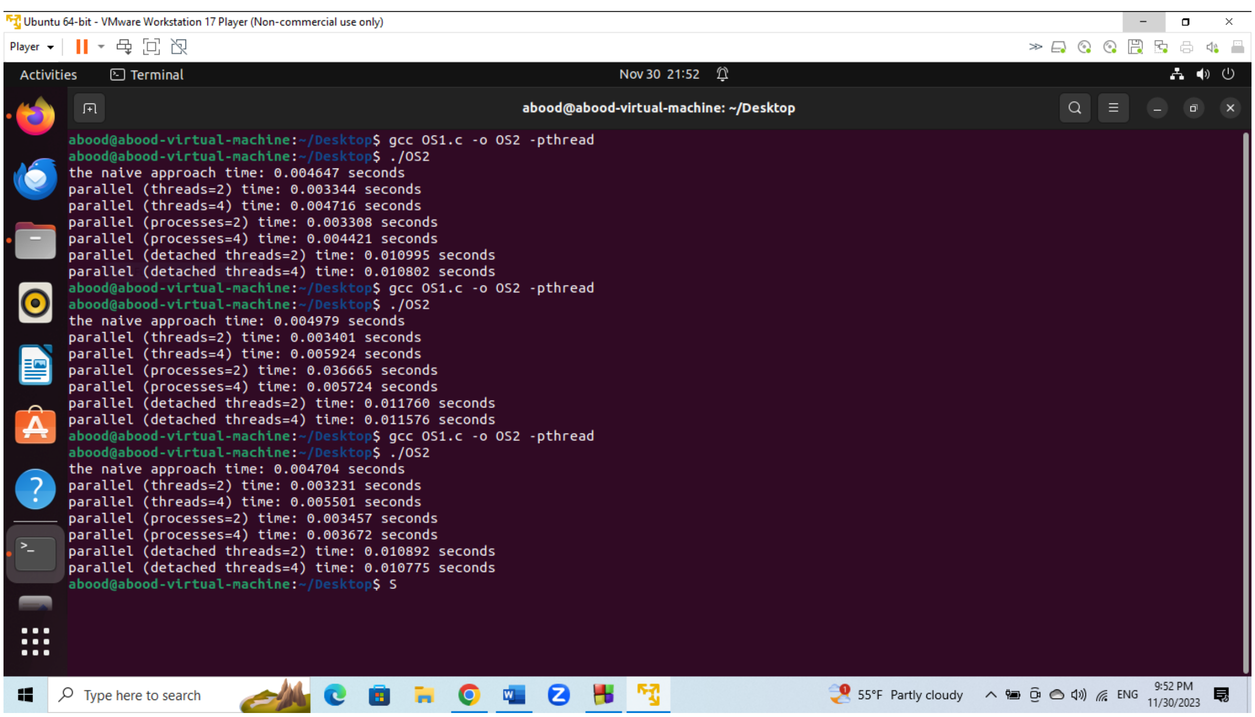
Matrix Multiplication for Parallel Processes: This loop uses the "parallel_m" function to multiply matrices for different numbers of processes (2, 4). It calculates how long each scenario takes and publishes the findings.

Matrix Multiplication for Detached Threads: This loop uses the "parallel_detache" function to do matrix multiplication for a variable number of detached threads (2, 4). It prints the findings and calculates the time needed for each example.

You may use the naive multiplication, threaded multiplication, parallel processes multiplication, and detachable threads multiplication methods in this code section to assess the efficacy of different matrix multiplication methods. It reports the duration of each approach using various setups.

THE RESULT:





Is it possible to measure the time in this approach? Yes, we can calculate the time because of the difference.

Exp	The Naive	Threads =2	Threads=4	Processes=2	Processes=4	Detached Threads=2	Detached Threads=4
1	0.004647	0.003344	0.004716	0.003308	0.004421	0.010995	0.010802
2	0.004979	0.003401	0.005924	0.036665	0.005724	0.011760	0.011576
3	0.004704	0.003231	0.005501	0.003457	0.003672	0.010892	0.010775

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

Ans: 4 child's or threads because I've 5 cores and it was easier to end the tasks