Operating Systems - Lab 2 Report

Team Members

- Amr Mohamed Elsayed 131

- Mohamed Ahmed Abdulraouf 152

- Mina Magued Mounir 222

- Youssef Hanii Salama 244

Part 1: Matrix Multithreaded Multiplication

- We had to make some sample Matrix instead of the one provided in main just for testing purposes

```
int A[X][Y] = {{1, 2, 3}, {4, 5, 6}};
int B[Y][Z] = {{7, 8}, {9, 10}, {11, 12}};
int C[X][Z];
```

- Implementing a single-threaded matrix multiplication was fairly easy, we implemented both the dot product and the single threaded function provided

- We have used the following structure

```
struct thread_data
{
    // You may need this fill this struct to pass and receive data
    // threads
    //output row;
    int row;
    //output column;
    int column;
    //value of element
    int value;

    // computed row
    int ret_row[X];
};
```

- We then defined two arrays of structs: One for element by element threaded multiplication and the other for row by row threaded multiplication

```
struct thread_data thread_data_array[X * Z];
struct thread_data thread_data_row_array[X];
```

- We fill these arrays using the data from the matrices inside these functions

```
fillTheArrayOfStructsWithData();
fillTheArrayOfArrStructsWithData();
```

We did though because the threads will run on a one-dimensional array, so we had to save the 2-dimensional data in a 1-dimensional form, which is represented by the struct arrays defined above in both cases

- Then we implemented dotProductThreadElem and dotProductThreadRow to calculate the dot product element-wise and row-wise respectively.
- Inside threadedMatMultPerElement()/perRow(), we first initlaize the struct array by calling the corresponding function from above, then we initialize an array of threads. We then run through that array and initlaize each thread separately using its offset from its corresponding struct array. After completing this, we join the threads together and they successfully complete the matrix multiplication task.
- On our sample matrix, A(2x3), B(3x2) => (2x2).
- In the element-wise threaded multiplication, we had 4 threads for our 2x2 matrix, because it contains four elements.
- In the row-wise threaded multiplication, we had 2 threads for our 2x2 matrix, because it contains two rows.

Sample run

```
Non-threaded C elements:
58
64
139
154
End of Non-threaded C elements
Time Spent in single-threaded: 0.000028



Element thread number = 0
Array value for row 0 column 0: 58
Element thread number = 2
Array value for row 1 column 0: 139
Element thread number = 3
Array value for row 1 column 1: 154
```

```
Element thread number = 1
Array value for row 0 column 1: 64
end of per element threading:
Time Spent in per-element-multi-threading : 0.000494



threadedMatMultPerRow function start
Row thread number = 0
Elements of C's row: 58
Elements of C's row: 64
Row thread number = 1
Elements of C's row: 139
Elements of C's row: 154
Time Spent in per-row-single-threaded: 0.000102
```

Performance metrics are quite random because they depend on the processor usage. We think that this might be because we're using a small example

---

Part 2: Water Reaction

- For this part, we created the following struct in reaction.h:

```
struct reaction {
    pthread_mutex_t lock;
    pthread_cond_t react, newH;
    int hCount;
};
```

- the struct contains one mutex(lock), two conditional variables (react and newH) and a counter for H atoms.
- Then we have written the initialization function that initializes the struct

```
void reaction_init(struct reaction *reaction)
{
    // clear the counters
    reaction->hCount = 0;

    // init the mutex
    pthread_mutex_init(&reaction->lock, 0);

    // init the condition variables
    pthread_cond_init(&reaction->react, 0);
    pthread_cond_init(&reaction->newH, 0);
}
```

- implementing reaction_h() a. lock critical section b. signal the creation of a new h after increasing the count - down by 1 c. block the reaction / down by one d. unlock critical section.

```
void reaction_h(struct reaction *reaction)
{
    // lock the critical section
    pthread_mutex_lock(&reaction->lock);

    ++reaction->hCount;
    // signal the creation of a new H
    // up by one
    pthread_cond_signal(&reaction->newH);
    // block the reaction
    // down by one
    pthread_cond_wait(&reaction->react, &reaction->lock);
```

```
    // realease access to the critical section
    pthread_mutex_unlock(&reaction->lock);
}
```

- implementing reaction_o() a. while not enough oxygen atoms -> wait and block reaction b. makeWater() -> when ready c. substract two from hCount // reset counter d. up by two signals e. unlock.

```
void reaction_o(struct reaction *reaction)
{
    pthread_mutex_lock(&reaction->lock);

    // block until there are 2 H atoms
    // down by one
    while (reaction->hCount < 2) pthread_cond_wait(&reaction->newH

    // when ready create water
    make_water();
    // reset the counter
    reaction->hCount -= 2;

    // unblock the reaction
    // up by two
    pthread_cond_signal(&reaction->react);
    pthread_cond_signal(&reaction->react);

    // release access to the critical section
    pthread_mutex_unlock(&reaction->lock);
}
```

Sample run

```
./reaction 0
Created 0 H and 200 O atoms (0.0% H), expecting 0 H2O molecules
```

```
Looks good!
./reaction 0
Created 0 H and 200 O atoms (0.0% H), expecting 0 H2O molecules
Looks good!
./reaction 20
Created 35 H and 165 O atoms (17.5% H), expecting 17 H2O molecules
Looks good!
./reaction 20
Created 40 H and 160 O atoms (20.0% H), expecting 20 H2O molecules
Looks good!
./reaction 40
Created 83 H and 117 O atoms (41.5% H), expecting 41 H2O molecules
Looks good!
./reaction 40
Created 81 H and 119 O atoms (40.5% H), expecting 40 H2O molecules
Looks good!
./reaction 60
Created 128 H and 72 O atoms (64.0% H), expecting 64 H2O molecules
Looks good!
./reaction 60
Created 120 H and 80 O atoms (60.0% H), expecting 60 H2O molecules
Looks good!
./reaction 80
Created 167 H and 33 O atoms (83.5% H), expecting 33 H2O molecules
Looks good!
./reaction 80
Created 161 H and 39 O atoms (80.5% H), expecting 39 H2O molecules
Looks good!
./reaction 100
Created 200 H and 0 O atoms (100.0% H), expecting 0 H2O molecules
Looks good!
./reaction 100
Created 200 H and 0 O atoms (100.0% H), expecting 0 H2O molecules
Looks good!
```