# OS LAB 2

## Multi-Threaded Matrix Multiplication

Made By:
Mohamed Ramadan Mohamed

# Problem Statement:

Implement a multi-threaded matrix multiplication using POSIX library threads interface. Using two methods: thread for each cell and thread for each row and compare time used.

# Code Organization:

program is divided into four main groups: 1- reading matrices from file. 2- launching threads for multiplication. 3-operating on matrices in each thread.4-some additional functions.

# Main Functions:

For group 1:

**readMat()**: this function is main function which is called by main to read everything from files. It opens the files and first reads the sizes for the two input matrices and validates that they can by multiplied. then allocates memory for both matrices and the two matrices for each method using **reserveMatrices()** which allocates each matrix using **reserveMatrixSize()**. Then it stores matrix elements from the file using load method for each matrix.

For group 2:

**execCell()**: a warp for **startCellMethod()** to calculate time and printing out information.
**execRow()**: the same as **execCell** for **startRowMethod()**.
**startCellMethod()**: this function loops N*M time and for every element in the resultant matrix it generate a thread to calculate its value.
**startRowMethod()**: this function loops N time for every row in the resultant matrix and generate a thread to calculate a row in it.

For group 3:

**calcCell()**: this function is used per thread in method 1 which takes the position of the cell to be calculated and calculates it then returns.
**calcRow():** this function is used per thread in method 2. it takes the row it will generate and calculate it then returns it.

For group 4:

**printOutput():** this function prints the desired matrix to the desired buffer.
**PrintTime():** this function prints the time difference between curTime1 and curTime2.

# Independence:

this code generated threads is independent which means that every thread operats on different isolated set of data so there's no need for access control like semaphore or mutex.

# Compiling Code:

1: open terminal using ctrl+alt+t.
2: change current working directory using “cd”.
3: type make clean.
4: type make
.. you're ready to run the program using ./matMultp

# Sample Runs:

1:
asmall:

```
row=1 col=1
1
```

bsmall:

```
row=1 col=2
1 2
```

command: `./matMultp asmall bsmall smallOut`

smallOut_1:

```
1.000000 2.000000
```

smallOut_2:

```
1.000000 2.000000
```

stdout:

```
------- ROW Method -------
Seconds taken: 0
Microseconds taken: 293
```

```
number of threads is: 1
------- Cell Method -------
Seconds taken: 0
Microseconds taken: 137

number of threads is: 2
```

2:

a_medium:

```
row=2 col=3
1 2 3
4 5 6
```

b_medium:

```
row=3 col=2
10 0.5
7 2.56
0.10 33
```

terminal:

```
./matMultp a_medium b_medium medium_out
------- ROW Method -------
Seconds taken: 0
Microseconds taken: 285

number of threads is: 2
------- Cell Method -------
Seconds taken: 0
Microseconds taken: 655

number of threads is: 4
```

medium_out_1:

```
24.300000    104.620000
75.600000    212.800000
```

medium_out_2:

```
24.300000    104.620000
75.600000    212.800000
```

3:

a_big:

```
row=5 col=4
   58.15960   26.72065   53.95446   42.57443
    1.82073    4.78339   27.66555    2.29948
   51.39746   44.07195   18.60176   48.86262
   29.79471   35.23627   16.51298   16.46449
    2.38234   39.49413    3.78693    0.50456
```

b_big:

```
row=4 col=6
   2.121509   1.717915   0.831141   0.791756   2.774805   3.310781
   2.358186   2.420850   5.581477   1.853777   5.954566   2.213915
   3.291562   4.354664   0.094657   1.745592   4.096855   0.974498
   5.677537   0.834988   2.812907   5.883498   0.665569   2.162273
```

terminal:

```
------- ROW Method -------
Seconds taken: 0
Microseconds taken: 518

number of threads is: 5
------- Cell Method -------
Seconds taken: 0
Microseconds taken: 1599

number of threads is: 30
```

big_out_1:

```
605.710729     435.102618     322.344601     440.251386
     569.871243     396.347000
119.261074     137.101942     37.298626 72.130661 148.407392
     48.550184
551.618209     316.792164     427.911904     442.348025
     513.777423     391.518987
294.134673     222.142635     269.310184     214.604047
     371.100744     228.346455
113.518233     116.614132     224.193378     84.678554 257.631255
     100.105405
```

big_out_2:

```
605.710729     435.102618     322.344601     440.251386
     569.871243     396.347000
119.261074     137.101942     37.298626 72.130661 148.407392
```

```
     48.550184
551.618209     316.792164     427.911904     442.348025
     513.777423     391.518987
294.134673     222.142635     269.310184     214.604047
     371.100744     228.346455
113.518233     116.614132     224.193378     84.678554 257.631255
     100.105405
```

4:
a.txt:
```
row=100 col=101
```
…

b.txt:
```
row=101 col=100
```
…

terminal:
```
./matMultp
------- ROW Method -------
Microseconds taken: 8216

number of threads is: 100
------- Cell Method -------
Microseconds taken: 147603

number of threads is: 10000
```

using diff for c_1 and c_2:
```
diff c_1 c_2
```
outputs no differences between two files

5: massive files
a1.in:
```
row=10000 col=1001
```
…

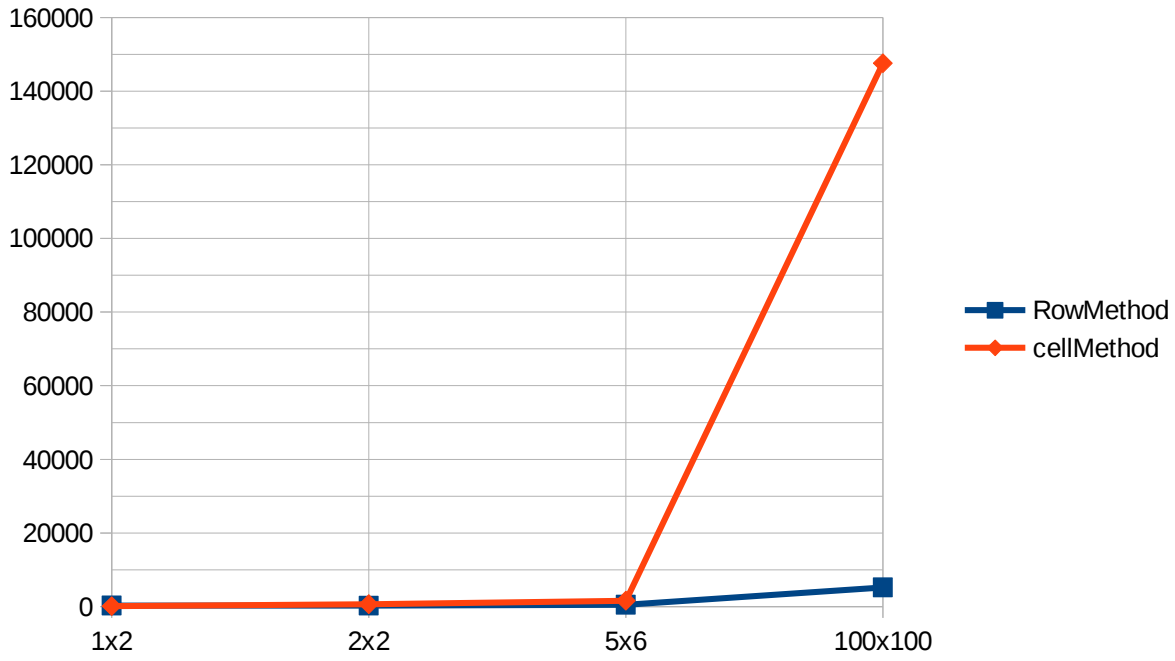
a2.in:
```
row=1001 col=1000
```
…

terminal:

```
------- ROW Method -------
Seconds taken: 20
Microseconds taken: 581080

number of threads is: 10000
------- Cell Method -------
cell thread creation error: Cannot allocate memory
```

# Comparison Between Times:




due to cost of creating threads in cell methods the time required to create a new thread is more than time required to solve the small problem