Akash Dansinghani
Homework2
Operating Systems
Professor Yan
02/14/2021

The five screenshots below are my code:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define NUM_THREADS 12

//#..

/*
Declare necessary global variables including 2D-array to store input and output matrices
int mat1[dim1][dim2] ...
Created variables mat1, mat2, mulMatrix with arrays that store the NUM_THREAD
 ...
 ...
*/
int mat1[NUM_THREADS][NUM_THREADS];
int mat2[NUM_THREADS][NUM_THREADS];
int mulMatrix[NUM_THREADS][NUM_THREADS];


/*
 Create data structure to store thread input info
 */
struct input {
  int i;  // tracks each index (row, column) in the matrix
  int *mat1;
  int (*mat2) [NUM_THREADS];
  int *mulMatrix;
};

/*
 Function Declaration:
void *mulMatrixMulter(void *params);
void ...
*/
void printRes(int res[NUM_THREADS][NUM_THREADS]);
```

```c
void printRes(int res[NUM_THREADS][NUM_THREADS]);
void *mat (void *param);
void readFile (char *s, int mat [NUM_THREADS][NUM_THREADS]);

int main(int argc, char *argv[]){

    /*
    Command prompt:
    ./hw2 mat1.txt mat2.txt
    */
  if(argc!=3){
    fprintf(stderr,"Usage: ./filename <mat1> <mat2>\n");
    return -1;
  }


  /**************
    1.Keep your code MODULARIZED and STRUCTURED!
        i.e. KEEP YOUR MAIN FUNCTION AS LIGHT AS NECESSARY!

    2. Pay attention to the data integrity when passing large collection data, e.g. array, list, queue, etc.

        Data coping of a large portion of matrix (e.g. entire row or column) or whole matrix
        should be avoided in your algorithm for multi-threading program.

     **************/

    // variable declarations
    int i, j, k, sum;

    pthread_t tid[NUM_THREADS]; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */
    pthread_attr_init(&attr); /* get the default attributes */

    printf("Read Matrix_1:\nRow: 12, col: 12\n");
    readFile("mat1.txt", mat1);
    printf("Read Matrix_2:\nRow: 12, col: 12\n");
```

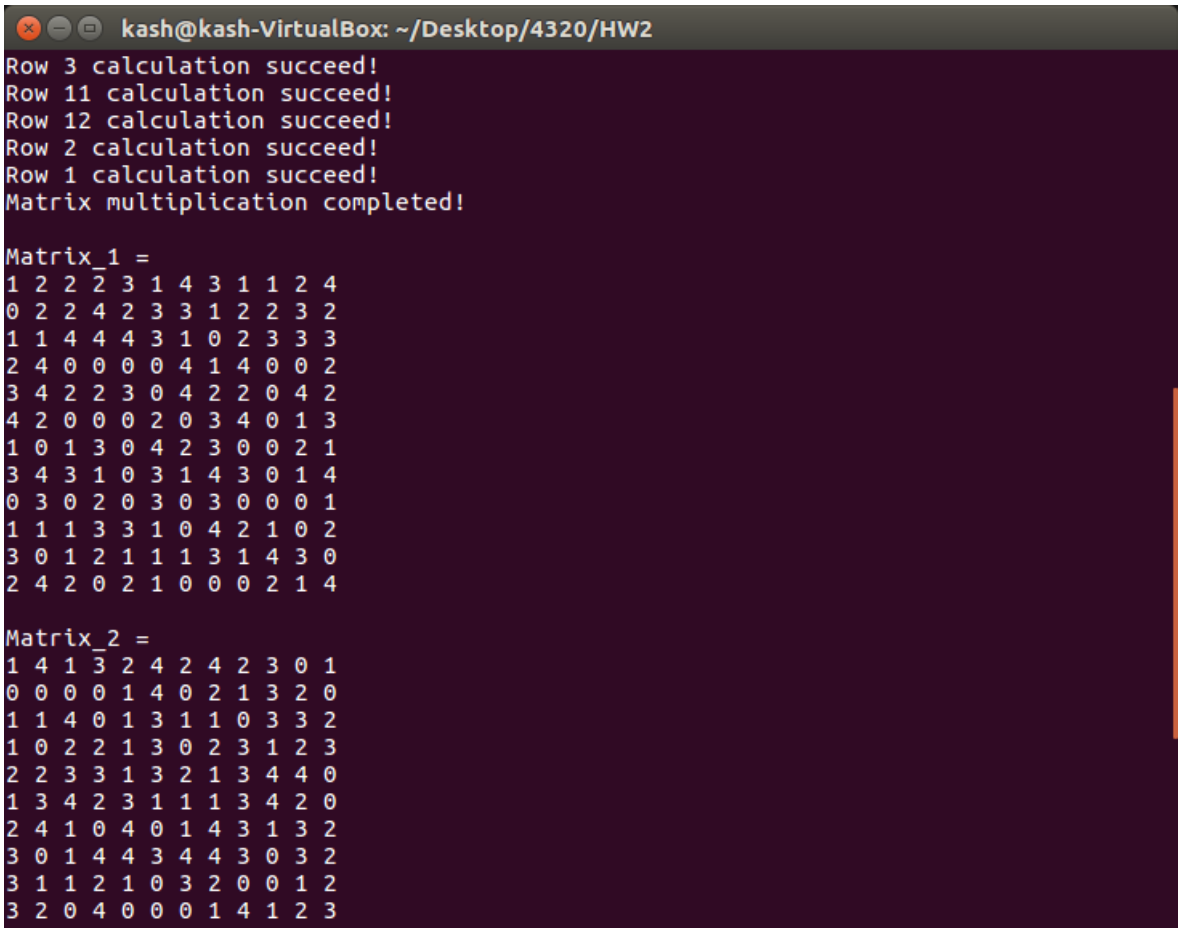
```c
    /**
     * information creating thread id and attribute vars below
     * https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032g/index.html

     **/

  // create structures for the threads arguments
  struct input args[NUM_THREADS];
  for (i = 0; i < NUM_THREADS; i++) {
    args[i].i = i + 1;
    args[i].mat1 = mat1[i];
    args[i].mat2 = mat2;
    args[i].mulMatrix = mulMatrix[i];
  }

  /* for loop to create each thread */
  for (i = 0; i < NUM_THREADS; i++) {
    printf("Creating thread no.%d\n", i);
    pthread_create(&(tid[i]),&attr, mat, &args[i]);
  }
  /* for loop to exit the creation of threads after going through all the rows */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(tid[i], NULL);
  }
  printf("Matrix multiplication completed!\n");

  printf("\nMatrix_1 = \n");
  printRes(mat1);
  printf("\nMatrix_2 = \n");
  printRes(mat2);
  printf("\nMatrix_1*Matrix_2 =\n");
  printRes(mulMatrix);


  return 0;
}
```

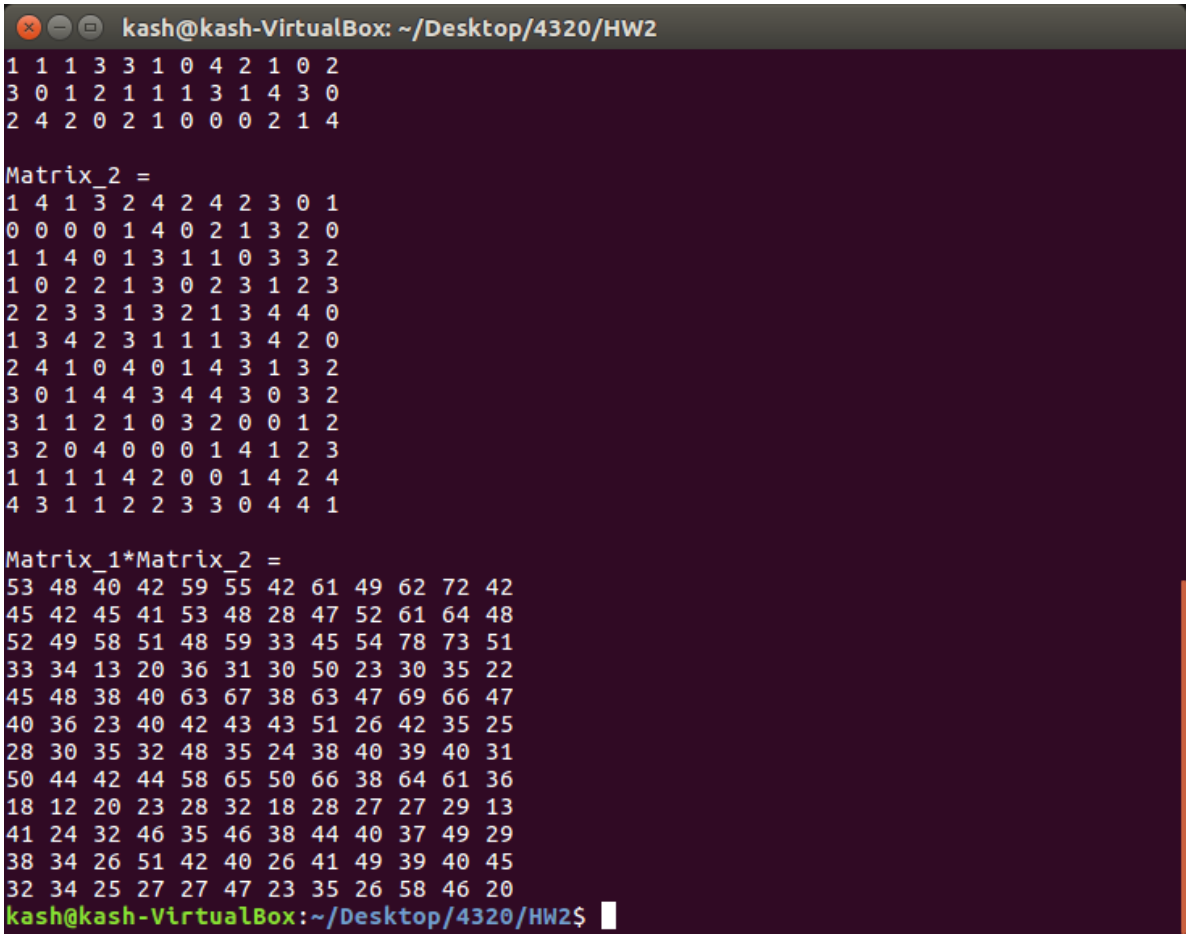
```c
/*
 Create your functions here..
*/

// function that takes in the matricies and multiplies the rows
void * mat (void *param) {
  struct input * input = (struct input*) param;
  int *mat1 = input->mat1;
  int *mulMatrix = input->mulMatrix;
  int (*mat2)[NUM_THREADS] = input->mat2;
  int i, j, k, sum;
      //calculate one row in the Matrix
      for (j = 0; j < NUM_THREADS; j++) {
        //calculate one element in the SMatrix
        sum = 0;
        for (i = 0; i < NUM_THREADS; i++) {
          sum += mat1[i] * mat2[i][j];
        }
        mulMatrix[j] = sum;
      }
  printf("Row %d calculation succeed!\n", input->i);
  pthread_exit(0);
}

// create the file as an input
void readFile (char *s, int mat [NUM_THREADS][NUM_THREADS]) {
    FILE* f;
    int index1, index2;
    //mat1
    if((f = fopen(s, "r")) == NULL)
        exit(1);

    for(index2=0; index2<NUM_THREADS; index2++)
        for(index1=0; index1<NUM_THREADS; index1++)
            if(fscanf(f, "%d", &mat[index2][index1]) != 1)
                exit(1);
    fclose(f);
```

```c
// print mulMatrix
void printRes(int res[NUM_THREADS][NUM_THREADS]){
    int index1, index2;
    for(index2=0; index2<NUM_THREADS; index2++){
        for(index1=0; index1<NUM_THREADS; index1++)
            printf ("%d ", res[index2][index1]);
        printf("\n");
    }
}
```

Commands to run the program and the output:

```
kash@kash-VirtualBox: ~/Desktop/4320/HW2
kash@kash-VirtualBox:~$ cd Desktop/
kash@kash-VirtualBox:~/Desktop$ cd 4320/
kash@kash-VirtualBox:~/Desktop/4320$ cd HW2
kash@kash-VirtualBox:~/Desktop/4320/HW2$ gcc -pthread hw2_AD.c -o hw2
kash@kash-VirtualBox:~/Desktop/4320/HW2$ ls
hw2         hw2.c        HW2_sc2.png  hw2_test.c  mat2.txt
hw2_AD.c    HW2sc1.png   HW2_sc3.png  mat1.txt
kash@kash-VirtualBox:~/Desktop/4320/HW2$ ./hw2 mat1.txt mat2.txt
Read Matrix_1:
Row: 12, col: 12
Read Matrix_2:
Row: 12, col: 12
Creating thread no.0
Creating thread no.1
Creating thread no.2
Creating thread no.3
Creating thread no.4
Creating thread no.5
Creating thread no.6
Creating thread no.7
Creating thread no.8
Creating thread no.9
Creating thread no.10
Creating thread no.11
Row 7 calculation succeed!
Row 6 calculation succeed!
Row 5 calculation succeed!
Row 8 calculation succeed!
Row 4 calculation succeed!
Row 9 calculation succeed!
Row 10 calculation succeed!
Row 3 calculation succeed!
```


kash@kash-VirtualBox: ~/Desktop/4320/HW2
Row 3 calculation succeed!
Row 11 calculation succeed!
Row 12 calculation succeed!
Row 2 calculation succeed!
Row 1 calculation succeed!
Matrix multiplication completed!
Matrix_1 =
1 2 2 2 3 1 4 3 1 1 2 4
0 2 2 4 2 3 3 1 2 2 3 2
1 1 4 4 4 3 1 0 2 3 3 3
2 4 0 0 0 0 4 1 4 0 0 2
3 4 2 2 3 0 4 2 2 0 4 2
4 2 0 0 0 2 0 3 4 0 1 3
1 0 1 3 0 4 2 3 0 0 2 1
3 4 3 1 0 3 1 4 3 0 1 4
0 3 0 2 0 3 0 3 0 0 0 1
1 1 1 3 3 1 0 4 2 1 0 2
3 0 1 2 1 1 1 3 1 4 3 0
2 4 2 0 2 1 0 0 0 2 1 4
Matrix_2 =
1 4 1 3 2 4 2 4 2 3 0 1
0 0 0 0 1 4 0 2 1 3 2 0
1 1 4 0 1 3 1 1 0 3 3 2
1 0 2 2 1 3 0 2 3 1 2 3
2 2 3 3 1 3 2 1 3 4 4 0
1 3 4 2 3 1 1 1 3 4 2 0
2 4 1 0 4 0 1 4 3 1 3 2
3 0 1 4 4 3 4 4 3 0 3 2
3 1 1 2 1 0 3 2 0 0 1 2
3 2 0 4 0 0 0 1 4 1 2 3


kash@kash-VirtualBox: ~/Desktop/4320/HW2
1 1 1 3 3 1 0 4 2 1 0 2
3 0 1 2 1 1 1 3 1 4 3 0
2 4 2 0 2 1 0 0 0 2 1 4
Matrix_2 =
1 4 1 3 2 4 2 4 2 3 0 1
0 0 0 0 1 4 0 2 1 3 2 0
1 1 4 0 1 3 1 1 0 3 3 2
1 0 2 2 1 3 0 2 3 1 2 3
2 2 3 3 1 3 2 1 3 4 4 0
1 3 4 2 3 1 1 1 3 4 2 0
2 4 1 0 4 0 1 4 3 1 3 2
3 0 1 4 4 3 4 4 3 0 3 2
3 1 1 2 1 0 3 2 0 0 1 2
3 2 0 4 0 0 0 1 4 1 2 3
1 1 1 1 4 2 0 0 1 4 2 4
4 3 1 1 2 2 3 3 0 4 4 1
Matrix_1*Matrix_2 =
53 48 40 42 59 55 42 61 49 62 72 42
45 42 45 41 53 48 28 47 52 61 64 48
52 49 58 51 48 59 33 45 54 78 73 51
33 34 13 20 36 31 30 50 23 30 35 22
45 48 38 40 63 67 38 63 47 69 66 47
40 36 23 40 42 43 43 51 26 42 35 25
28 30 35 32 48 35 24 38 40 39 40 31
50 44 42 44 58 65 50 66 38 64 61 36
18 12 20 23 28 32 18 28 27 27 29 13
41 24 32 46 35 46 38 44 40 37 49 29
38 34 26 51 42 40 26 41 49 39 40 45
32 34 25 27 27 47 23 35 26 58 46 20
kash@kash-VirtualBox:~/Desktop/4320/HW2$

A high-level description of major components/functionality:

```
int mat1[NUM_THREADS][NUM_THREADS];
int mat2[NUM_THREADS][NUM_THREADS];
int mulMatrix[NUM_THREADS][NUM_THREADS];
```

Above we can see I initialized three arrays, one called “mat1”, next “mat2”, and last the final matrix called “mulMatrix” which stored the threads by using “NUM_THREADS” embedded in each array. This allows us to execute multiple process using parallel proccesing to facilitate multiplication between two matrices and storing it into the final matrix.

```
void printRes(int res[NUM_THREADS][NUM_THREADS]);
void *mat (void *param);
void readFile (char *s, int mat [NUM_THREADS][NUM_THREADS]);
```

Then I created functions to open mat1 and mat2 and stored it in the 2D array. The pointer *mat is used to work through the multiplications between the matrices efficiently We can see below, the function “readFile” takes in the txt file called “mat1.txt” and stores it into the variable we declared “mat1” and the same goes for “mat2”.

```
// variable declarations
int i, j, k, sum;

pthread_t tid[NUM_THREADS]; /* the thread identifier */
pthread_attr_t attr; /* set of thread attributes */
pthread_attr_init(&attr); /* get the default attributes */

printf("Read Matrix_1:\nRow: 12, col: 12\n");
readFile("mat1.txt", mat1);
printf("Read Matrix_2:\nRow: 12, col: 12\n");
readFile("mat2.txt", mat2);
```

Next, I used pthread (API to manage threads) as an identifier, attribute, default attribute which was taught in lecture 4 about threads (Slide 25). Using pthread_t I was able to check if I was able to multiply two values in each of the respective matrices since it is able to utilize threads to do multiple processes at once. Once successful I was able to multiply each index between the matrices.

```c
void * mat (void *param) {
  struct input * input = (struct input*) param;
  int *mat1 = input->mat1;
  int *mulMatrix = input->mulMatrix;
  int (*mat2)[NUM_THREADS] = input->mat2;
  int i, j, k, sum;
      //calculate one row in the Matrix
      for (j = 0; j < NUM_THREADS; j++) {
        sum = 0;
        //calculate one element in the Matrix
        for (i = 0; i < NUM_THREADS; i++) {
          sum += mat1[i] * mat2[i][j];
        }
        mulMatrix[j] = sum;
      }
  printf("Row %d calculation succeed!\n", input->i);
  if (j || i > 12){
    pthread_exit(0);
  }
}
```

I used function above which calculates one row in a matrix and calculates one element in the matrix and then adds the values and stores it into the multiplied matrix variable called mulMatrix.

```
Row 7 calculation succeed!
Row 6 calculation succeed!
Row 5 calculation succeed!
Row 8 calculation succeed!
Row 4 calculation succeed!
Row 9 calculation succeed!
```

```
Row 3 calculation succeed!
Row 11 calculation succeed!
Row 12 calculation succeed!
Row 2 calculation succeed!
Row 1 calculation succeed!
Matrix multiplication completed!
```

As we can see above, the function calculates each row and element of both matrices and stores them into mulMatrix from the value in sum. Each matrix has a ptr (pointer) which we are able to call them by their name by using “input -> *variable*. Since there are pointers for each matrix, we are able to perform each operation of multiplication between each matrix row and element and store each value in the mulMatrix variable.

Once all rows are calculated, the values are stored in the mulMatrix pointer and I used that to pass any arguments in the rest of my code.

```
/* for loop to exit the creation of threads after going through all the rows */
for (i = 0; i < NUM_THREADS; i++) {
  pthread_join(tid[i], NULL);
}
printf("Matrix multiplication completed!\n");
```

Created a for loop that will exit the multiplication process after there are no more elements to calculate.
After all this is done, we can print out the result with the printRes class below

```
void printRes(int a[NUM_THREADS][NUM_THREADS]){
    int index1, index2;
    for(index2=0; index2<NUM_THREADS; index2++){
        for(index1=0; index1<NUM_THREADS; index1++)
            printf ("%d ", a[index2][index1]);
        printf("\n");
```

Discussion on the Pros and Cons regarding efficiency and cost when:

Using 1 thread:
Pro - It is more efficient in using space (using less memory)
Con - More runtime (takes more time to execute the program for multiplying matrices)
Using 12 threads:
Pro - Faster than using < 12 threads
Con - Requires more storage (memory) than < 12 threads.
Using 144 threads:
Pro - Fastest out of the three options. It will run faster if program compare is < 144 threads
Con: Slowest out of the three options. It requires more storage than the other programs given they use < 144 threads.
Ultimately we can see that the greater the amount of threads used, the faster the process but uses more space. We want to use less threads if the program is simple or does not have multiple processes to go through because we do not need to use lots of space and the runtime will still be relatively fast. We should use more threads if the program is more complex.