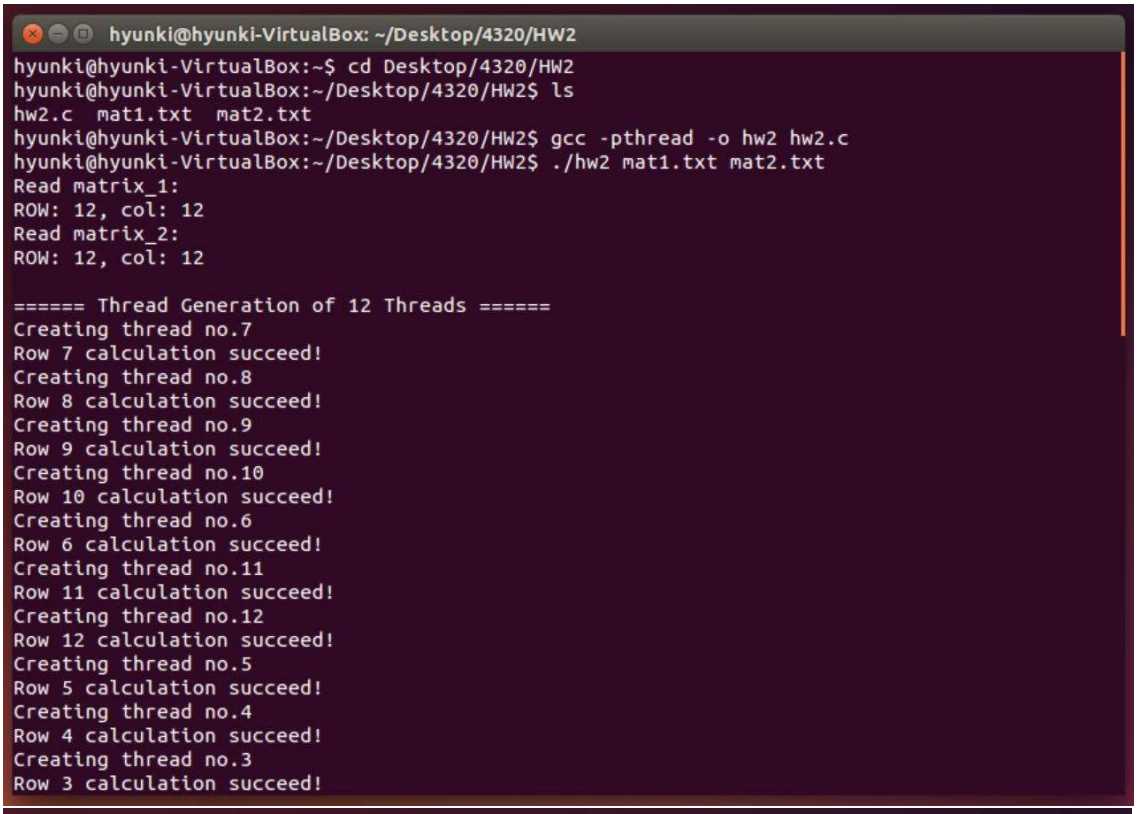
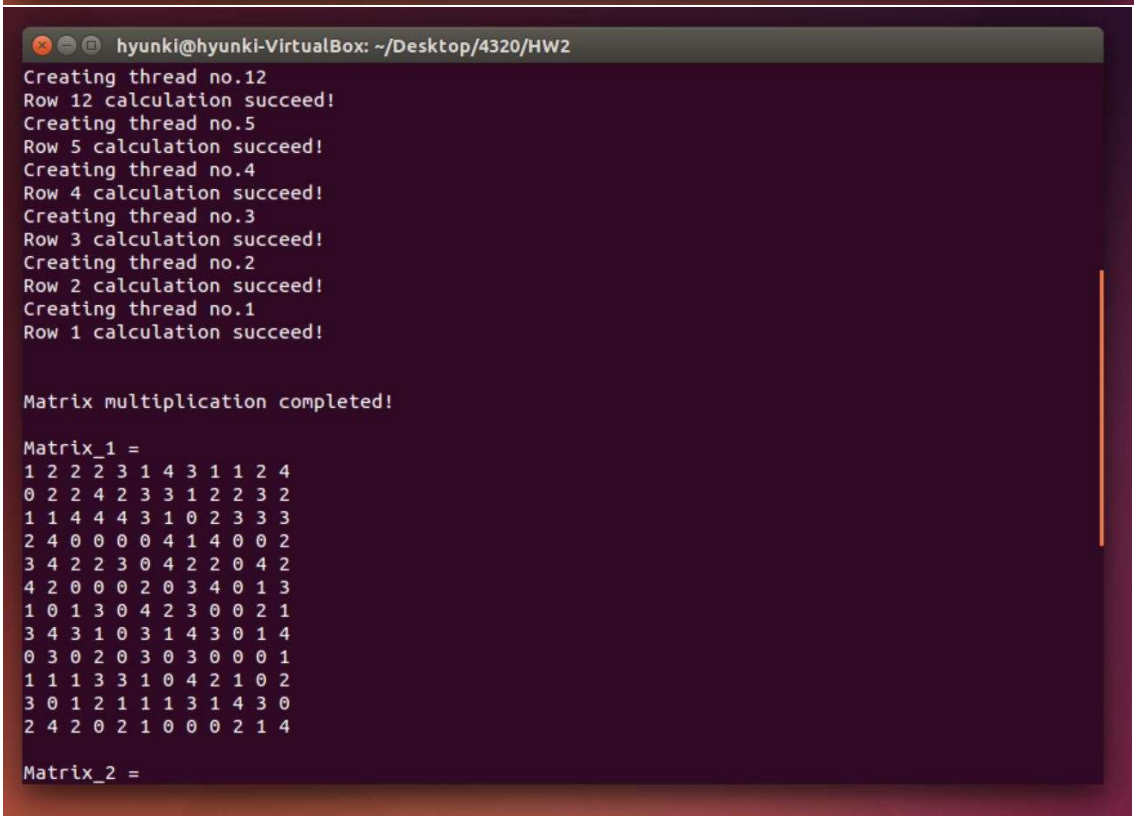
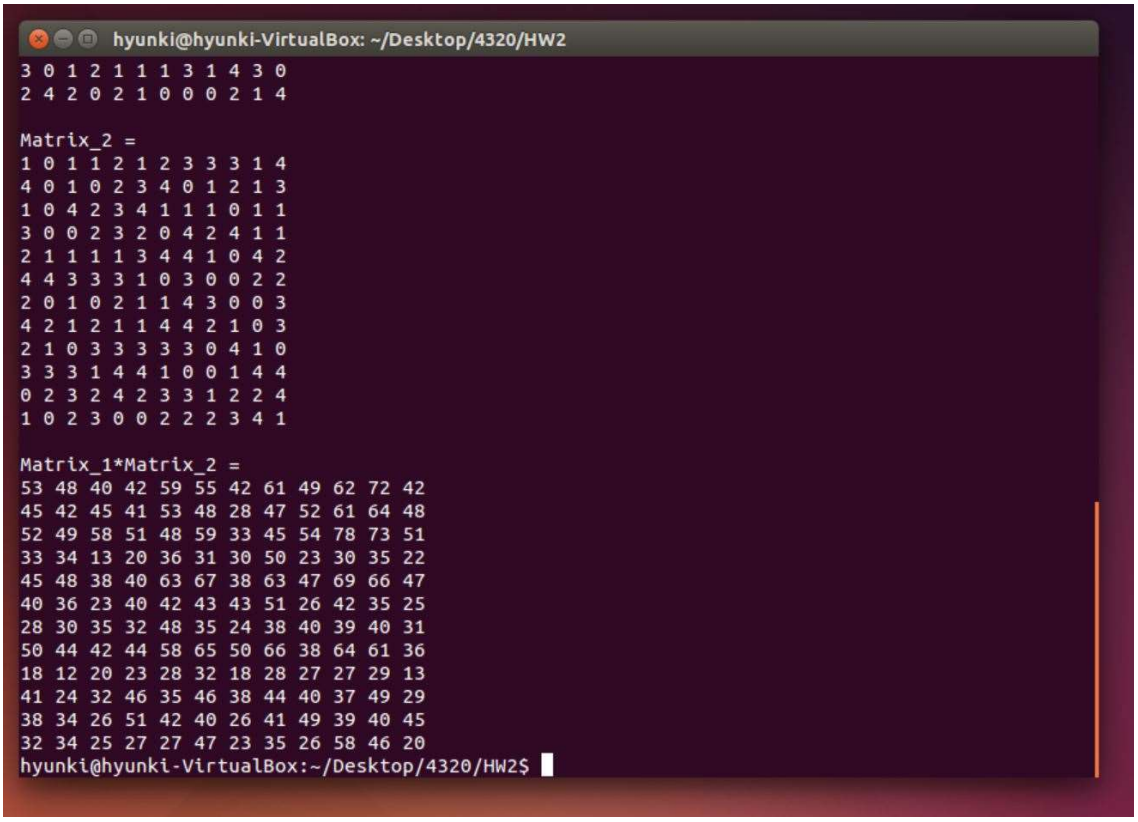
HomeWork2
CSC4320

#002-34-4677
Hyunki Lee

1. Screenshots of output







2. Major components/functionality

   a) Matrix Multiplication

   The rows on the first matrix and the columns of the second matrix will be multiplied. The function of multiplication considers the multiply order that multiplying rows by columns.

   b) Pthreads

   The one of main purpose of this assignment is to calculate matrix factors with different threads. Each thread will calculate each matrix calculation. First, defined structure has matrices information. It means that the structure hold matrices factors such as number of rows, matrix pointers. The reason is that multiple threads can exist within the same process thus they can share resources. Threads will be created by pthread create(): pthread_create(pthread_t *thread_id, const pthread_attr_t *attributes, void *(*thread_function)(void *), void *arguments). Pthread t is an opaque type, and this is acting for new threads. Attribute are used thread creation time for the new threads. If the value is NULL, the default attributes will be used. Thread function part is that the new threads will be executing. In this assignment,

thread function part is the actual multiplication for matrices. The void argument is that the pointer is passed as an argument to thread function. Because of this argument, we need to hold matrix information in a structure. Pthreads can terminate to use pthread_exit(void *return). Return is the return value of a thread. Pthread_join: this command makes that a thread waits for other threads to terminate. For Pthread_join, thread_t thread and status ptr is needed.

c) Matrix multiplication with multi threads.

Important factors to accomplish this assignment are a appropriate structure, position of pthread_create command, and function for matrix multiplication. Twelve threads are used. Each thread should get matrix information from a structure. Using pointers from the structure and for loop (twelve times) will give matrix information to each thread, and the threads will pass the matrix multiplication function. Each thread passes the function, which is multiplying rows by columns, and exit the function. The twelve threads will calculate each row thus using pthread_join to wait until all threads execute the function. Even though one thread computes one row, the threads share a memory. Therefore, we can read the result like one process.

3. Discussion

a) Using 1 thread

If we use 1 thread, we do not need to consider synchronization of threads. For instance, which thread should access which values or resource, and which should not. Also, if the thread has an error, we need to fix only one thread. Therefore, one thread can reduce cost for maintain the program. However, if we have enough resource such as multi CPU to run this program, using only one thread cannot utilize the resources. We can say using one thread is less efficiency.

b) Using 12 threads

This is the most efficiency case for this assignment if we have enough resources for running 12 threads. We need twelve multiplication, and we have twelve threads. It means all threads work. Creating new thread is faster than create new process. Otherwise, if one thread has an error the whole process will not operate correctly. Furthermore, one of efficiency is sharing resources, but using amount of resources is not efficient.

c) Using 144threads

In this case, if we have enough to run the 144 threads, we may have fastest result from the program. On the other hands, if we do not have enough resources, the performance

will be result as performance degradation. When threads use CPU, the context switching will occur. This process takes a lot of costs. We should take care of each thread because if one thread does not work, the whole process does not work. When one or more threads work, other threads may not work (waiting). If some threads waiting for their turn, the process would be delayed than using less threads. Therefore, using 144 threads for the assignment is not efficiency.

4. Copy of C source code

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

#define NUM_THREADS 12
#define MAT_SIZE 12

int mat1[MAT_SIZE][MAT_SIZE];
int mat2[MAT_SIZE][MAT_SIZE];
int ret[MAT_SIZE][MAT_SIZE];

void printAllMatrices();

//multiplication function
void *matrixMulter(void *params);

//information for matrices
struct ArgStructure {
        int rowNum;
        int *mat1Ptr;
        int *mat2Ptr;
        int *retPtr;
};
```

```
int main(int argc, char *argv[]) {
	/*
	  Command prompt:
	  ./hw2 mat1.txt mat2.txt
	*/

		int x,y,i;
		if (argc != 3) {
			fprintf(stderr, "Usage: ./filename <mat1> <mat2>\n");
			return -1;
		}

		// Read matrices
		FILE *fp1 = fopen(argv[1], "r");
		printf("Read matrix_1:\nROW: %d, col: %d\n", MAT_SIZE, MAT_SIZE);
		FILE *fp2 = fopen(argv[2], "r");
		printf("Read matrix_2:\nROW: %d, col: %d\n", MAT_SIZE, MAT_SIZE);
		printf("\n");

		// Copy matrices
		for (x = 0; x < MAT_SIZE; x++) {
			for (y = 0; y < MAT_SIZE; y++) {
				fscanf(fp1, "%d", &mat1[x][y]);
				// mat2 need to be flipped for multiply
				fscanf(fp2, "%d", &mat2[y][x]);
			}
		}
		// Close files
		fclose(fp1);
```

```
	fclose(fp2);


	// Thread generation
	printf("====== Thread Generation of %d Threads ======\n", NUM_THREADS);
	pthread_t tid[NUM_THREADS];
	//create threads
	for (i = 0; i < NUM_THREADS; i++) {
		struct ArgStructure *argStruct;
		argStruct = (struct ArgStructure *) malloc(sizeof(struct ArgStructure));
		(*argStruct).rowNum = i;
		(*argStruct).mat1Ptr = *mat1;
		(*argStruct).mat2Ptr = *mat2;
		(*argStruct).retPtr = *ret;
		pthread_create(&tid[i], NULL,  matrixMulter, (void *) argStruct);
	}
	// threads finished operating
	for (i = 0; i < NUM_THREADS; i++) {
		pthread_join(tid[i], NULL);
	}
		printf("\n\n");
		printf("Matrix multiplication completed!\n\n");


		printAllMatrices();


		return 0;
}
//multiplication function
void *matrixMulter(void *arguments) {
	int col;
```

```
        int subCol;
        struct ArgStructure *args = (struct ArgStructure*)arguments;
        int rowNum = (*args).rowNum;
        int (*mat1Ptr) = (*args).mat1Ptr+(rowNum*MAT_SIZE);
        int (*mat2Ptr) = (*args).mat2Ptr;
        int (*retPtr) = (*args).retPtr+(rowNum*MAT_SIZE);
        printf("Creating thread no.%d\n", rowNum+1);

        for (col = 0; col < MAT_SIZE; col++) {

                int multiplyResult = 0;
                for (subCol = 0; subCol < MAT_SIZE; subCol++) {
                        int mat1Val = *(mat1Ptr+subCol);
                        int mat2Val = *(mat2Ptr+(col*MAT_SIZE)+subCol);
                        multiplyResult += (mat1Val * mat2Val);
                }
          *(retPtr+col) = multiplyResult;
        }

        printf("Row %d calculation succeed!\n", rowNum+1);
        free(args);
        pthread_exit((void *) 0);
}
// print matrix function
void printAllMatrices() {
        int x,y;
        printf("Matrix_1 = \n");
        for (x = 0; x < MAT_SIZE; x++) {
                for (y = 0; y < MAT_SIZE; y++) {
                        printf("%d ", mat1[x][y]);
```

```
            }
        printf("\n");
    }
    printf("\n");

    printf("Matrix_2 = \n");
    for (x = 0; x < MAT_SIZE; x++) {
            for (y = 0; y < MAT_SIZE; y++) {
                    printf("%d ", mat2[x][y]);
            }
        printf("\n");
    }
    printf("\n");

    printf("Matrix_1*Matrix_2 = \n");
    for (x = 0; x < MAT_SIZE; x++) {
            for (y = 0; y < MAT_SIZE; y++) {
                    printf("%d ", ret[x][y]);
            }
        printf("\n");
    }
}
```