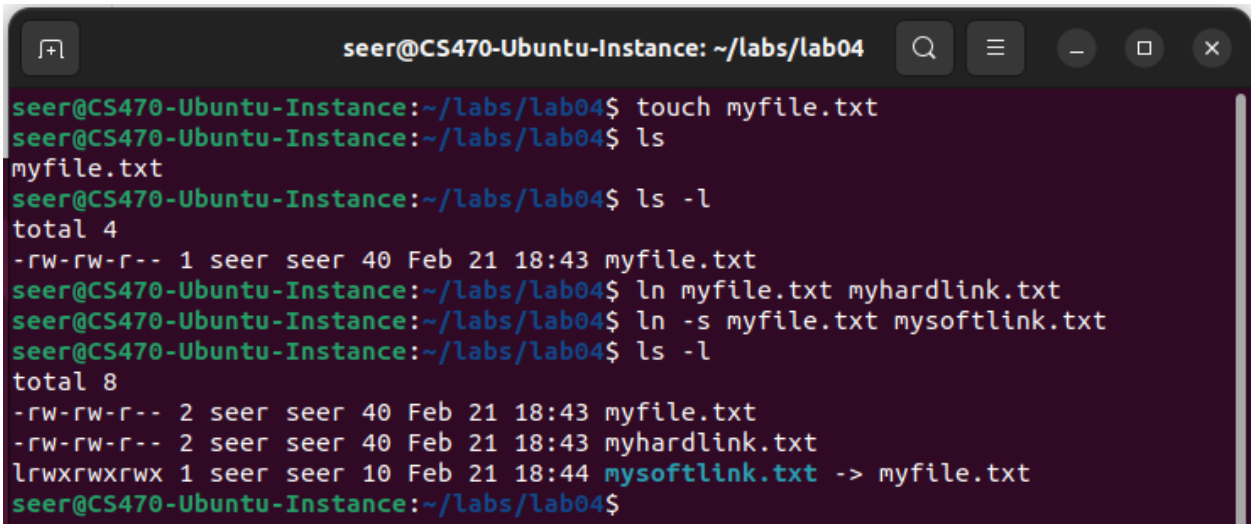
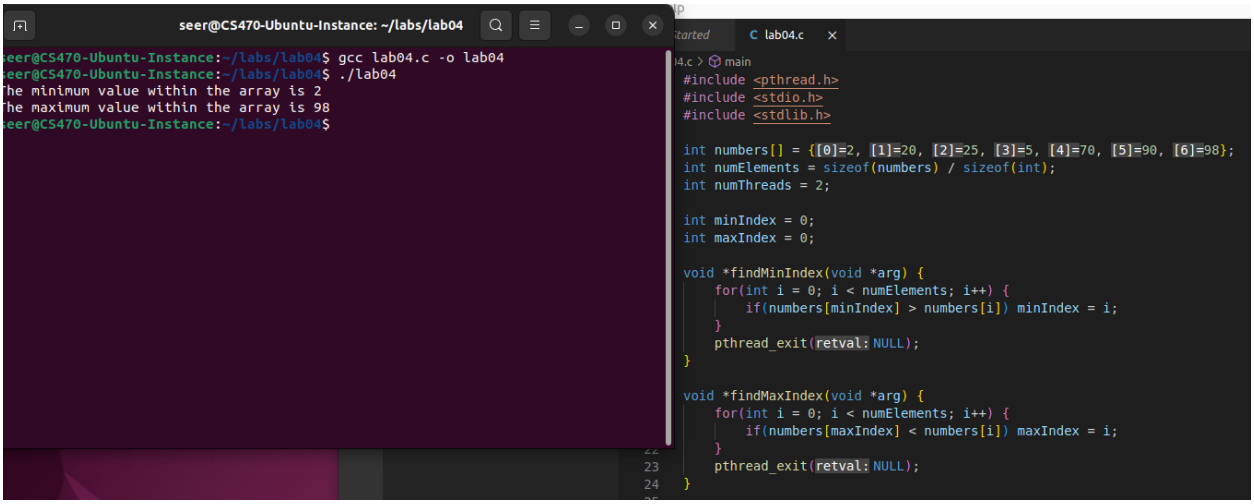
1. Using the ln and ln -s commands, create hard and soft links.



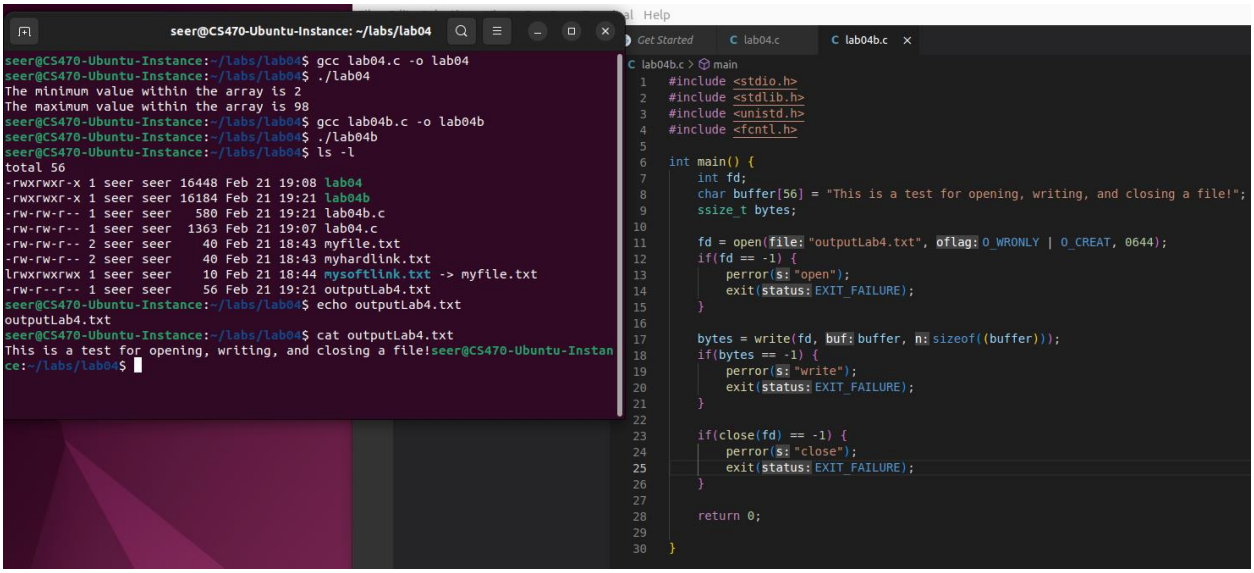
2. Create a multithreaded program that computes different statistical values for a set of numbers.



```c
int main() {

    pthread_t threads[numThreads];
    int returnCode;

    returnCode = pthread_create(newthread: &threads[0], attr: NULL, start_routine: findMinIndex, arg: NULL);
    if(returnCode) {
        printf(format: "Error: Unable to create thread.\n");
        exit(status: -1);
    }

    returnCode = pthread_create(newthread: &threads[1], attr: NULL, start_routine: findMaxIndex, arg: NULL);
    if(returnCode) {
        printf(format: "Error: Unable to create thread.\n");
        exit(status: -1);
    }

    for(int i = 0; i < numThreads; i++) {
        returnCode = pthread_join(th: threads[i], thread_return: NULL);
        if(returnCode) {
            printf(format: "Error: Unable to join threads.\n");
            exit(status: -1);
        }
    }

    printf(format: "The minimum value within the array is %d\n", numbers[minIndex]);
    printf(format: "The maximum value within the array is %d\n", numbers[maxIndex]);
    pthread_exit(retval: NULL);
    return 0;
}
```

The above code utilizes multithreading to find the indices of the minimum and maximum values within the array, spread over two loops. The minimum is found by comparing the value within the index with each other value, and if a smaller value is found, the minimum index is replaced. Similarly, the maximum is found using a similar principle, except comparing against larger values. If a larger value exists, then the maxIndex is set to the index where the higher value was found. Next, the threads are created using pthread_create, and any errors are logged if the threads cannot be created. Finally, the threads are joined back together into the main process once both the minimum and maximum values have been found.

3. Write a C program that opens the file “outputLab4.txt” for writing and appends the phrase “This is a test for opening, writing, and closing a file!”



This is a singularly threaded process that creates a file called “outputLab4.txt” in the local folder, writes to it, and finally closes it. The variable “fd” stands for file data, or file descriptor, which contains the current status of the file. “bytes” stores the the number of bytes written to the file, and “buffer” stores a 56 character buffer that contains the text to be written into the file. Each step has a verification check to ensure that the file has been properly written to the operating system, including the closing of the program.

4. Write a program for matrix addition, subtraction, and multiplication using multithreading.

```
seer@CS470-Ubuntu-Instance:~/labs/lab04$ gcc lab04c.c -o lab04c
seer@CS470-Ubuntu-Instance:~/labs/lab04$ ./lab04c
Matrix sum:
6, 6, 4, 11,
8, 5, 11, 12,
5, -8, 1, 7,
10, 8, 13, 16,
Matrix difference:
4, 0, 0, 3,
4, -3, -1, 6,
-9, -2, 5, 9,
2, 2, 5, -4,
Matrix product:
5, 9, 4, 28,
12, 4, 30, 27,
-14, 15, -6, -8,
24, 15, 36, 60,
seer@CS470-Ubuntu-Instance:~/labs/lab04$
```

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


int matrixA[4][4] = {
    [0]={[0]=5, [1]=3, [2]=2, [3]=7},
    [1]={[0]=6, [1]=1, [2]=5, [3]=9},
    [2]={ [0]=-2, [1]=-5, [2]=3, [3]=8},
    [3]={[0]=6, [1]=5, [2]=9, [3]=6}
};

int matrixB[4][4] = {
    [0]={[0]=1, [1]=3, [2]=2, [3]=4},
    [1]={[0]=2, [1]=4, [2]=6, [3]=3},
    [2]={[0]=7, [1]=-3, [2]=-2, [3]=-1},
    [3]={[0]=4, [1]=3, [2]=4, [3]=10}
};

int matrixSum[4][4], matrixDiff[4][4], matrixProd[4][4];
int numThreads = 3;

void *matrixAddition(void *arg) {
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            matrixSum[i][j] = matrixA[i][j] + matrixB[i][j];
        }
    }
    printf(format: "Matrix sum:\n");
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            printf(format: "%d, ", matrixSum[i][j]);
        }
        printf(format: "\n");
    }
    pthread_exit(retval: NULL);
}
```

```c
void *matrixSubtract(void *arg) {
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            matrixDiff[i][j] = matrixA[i][j] - matrixB[i][j];
        }
    }
    printf(format: "Matrix difference:\n");
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            printf(format: "%d, ", matrixDiff[i][j]);
        }
        printf(format: "\n");
    }
    pthread_exit(retval: NULL);
}

void *matrixMultiply(void *arg) {
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            matrixProd[i][j] = matrixA[i][j] * matrixB[i][j];
        }
    }
    printf(format: "Matrix product:\n");
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            printf(format: "%d, ", matrixProd[i][j]);
        }
        printf(format: "\n");
    }
    pthread_exit(retval: NULL);
}
```

```c
int main() {

    pthread_t threads[numThreads];
    int returnCode;

    returnCode = pthread_create(newthread: &threads[0], attr: NULL, start_routine: matrixAddition, arg: NULL);
    if(returnCode) {
        printf(format: "Error: Unable to create thread.\n");
        exit(status: -1);
    }

    returnCode = pthread_create(newthread: &threads[1], attr: NULL, start_routine: matrixSubtract, arg: NULL);
    if(returnCode) {
        printf(format: "Error: Unable to create thread.\n");
        exit(status: -1);
    }

    returnCode = pthread_create(newthread: &threads[2], attr: NULL, start_routine: matrixMultiply, arg: NULL);
    if(returnCode) {
        printf(format: "Error: Unable to create thread.\n");
        exit(status: -1);
    }

    for(int i = 0; i < numThreads; i++) {
        returnCode = pthread_join(th: threads[i], thread_return: NULL);
        if(returnCode) {
            printf(format: "Error: Unable to join threads.\n");
            exit(status: -1);
        }
    }

    pthread_exit(retval: NULL);
    return 0;

}
```

This is a multi-threaded program designed to perform addition, subtraction, and multiplication separately on two matrices. The two matrices, MatrixA and MatrixB, are created with a set of arbitrary values. The functions “matrixAddition”, “matrixSubstract”, and “matrixMultiply” are multithreading-compatible functions that each respectively add, subtract, and multiply two matrices with eachother in O(n^2) time. In the main function, the three threads are created, and are verified that they have been created, with each thread tasked with accomplishing a specific function. Finally, after all threads have completed their jobs, the threads are rejoined into the main thread, and the program is exited safely.