Fiona Le

*Date: 03/14/2021*

**I. Code:**

**#include <stdio.h>**

**#include <pthread.h>**

**#include <stdlib.h>**

**/\*\*\*\*\* GLOBAL VARIABLES \*\*\*\*\*/**

**// Mutex variable to control which thread can access to the shared variables**

**pthread\_mutex\_t mutex;**

**// An original array - an provided array from user**

**int\* unsortedArray;**

**// An sorted array**

**int\* sortedArray;**

**// Size of an array**

**int arraySize;**

**// Index of the Threads**

**int threadID = 0;**

**/\*\*\*\*\* FUNCTIONS \*\*\*\*\*/**

**void\* sorter(void \*params);**

**void\* merger(void\* params);**

**/\*\*\*\*\* MAIN FUNCTION \*\*\*\*\*/**

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

**{**

**/\*\*\*\*\* INPUT \*\*\*\*\*/**

**// Get the size of an array**

**printf("Enter the size of an array: ");**

**scanf("%d", &arraySize);**

**// Generate an unsortedArray and sortedArray**

**unsortedArray = (int\*)calloc(arraySize, sizeof(int));**

**sortedArray = (int\*)calloc(arraySize, sizeof(int));**

**// Get an array from user**

**printf("Enter an array: ");**

**for (int i = 0; i < arraySize; i++)**

**scanf("%d", &unsortedArray[i]);**

**/\*\*\*\*\* CREATE THREADS \*\*\*\*\*/**

**// Create 3 threads in total: 2 sorting threads and 1 merging threads**

**pthread\_t threads[3];**

**// Create a mutex**

**pthread\_mutex\_init(&mutex, NULL);**

**/\*\*\*\*\* SPLIT AN ARRAY INTO 2 PARTS THEN SORT \*\*\*\*\*/**

**for (int i = 0; i < 2; i++)**

**pthread\_create(&threads[i], NULL, sorter, (void\*) unsortedArray);**

**for (int i = 0; i < 2; i++)**

**pthread\_join(threads[i], NULL);**

**/\*\*\*\*\* MERGE 2 PARTS INTO ONE ARRAY \*\*\*\*\*/**

**pthread\_create(&threads[3], NULL, merger, (void\*) unsortedArray);**

**pthread\_join(threads[3], NULL);**

**/\*\*\*\*\* PRINT AN ARRAY AFTER SORTING \*\*\*\*\*/**

**printf("AN ARRAY AFTER SORTING: ");**

**for (int i = 0; i < arraySize; i++)**

**printf("%d ", sortedArray[i]);**

**printf("\n");**

**/\*\*\*\*\* EXIT THREADS & DESTROY MUTEX \*\*\*\*\*/**

**pthread\_exit(NULL);**

**pthread\_mutex\_destroy(&mutex);**

**return EXIT\_SUCCESS;**

**}**

**/\*\*\*\*\* Sorting Thread \*\*\*\*/**

**void\* sorter(void \*params)**

**/\***

**+ Introduction: Slipt an array unsortedArray into 2 parts and sort each part.**

**+ Input:**

**\* unsortedArray**

**\* mutex: to lock/unlock the right of accessing shared Data**

**\* threadID: ID of the current thread**

**\* arraySize: size of an array**

**+ Output: unsortedArray with 2 parts are sorted.**

**\*/**

**{**

**// Lock a mutex**

**pthread\_mutex\_lock(&mutex);**

**// Initialize needed variables for this function**

**int\* array = (int\*)params;**

**int temp, r, l;**

**// Indetify the starting and finishing elements for each part to begin sorting**

**threadID++;**

**// Thread 1 will sort the elements from 0 to arraySize / 2 - 1**

**if (threadID == 1)**

**{**

**l = 0;**

**r = arraySize / 2;**

**}**

**// Thread 2 will sort the elements from arraySize / 2 to arraySize**

**else**

**{**

**l = arraySize / 2;**

**r = arraySize;**

**}**

**// Selection Sort**

**for (int i = l; i < r - 1; i++)**

**for (int j = i + 1; j < r; j++)**

**if (array[j] < array[i])**

**{**

**temp = array[i];**

**array[i] = array[j];**

**array[j] = temp;**

**}**

**// Copy the sorted part into the provided array**

**for (int i = l; i < r; i++)**

**unsortedArray[i] = array[i];**

**// Release a mutex**

**pthread\_mutex\_unlock(&mutex);**

**// Exit thread**

**pthread\_exit(NULL);**

**}**

**/\*\*\*\*\* Merging Thread \*\*\*\*\*/**

**void\* merger(void\* params)**

**/\***

**+ Introduction: Merge two ordered sub-arrays of unsortedArray into one ascending sequence called sortedArray.**

**+ Input:**

**\* unsortedArray**

**\* arraySize: size of an array**

**+ Output: sortedArray**

**\*/**

**{**

**// Initialize needed variables for this function**

**int \* array = (int\*)params;**

**int index = 0;**

**int l1 = 0, l2 = arraySize / 2, r1 = arraySize / 2, r2 = arraySize;**

**/\* Merge array:**

**(1) Go through each element in each part and compare it with the element of another part to consider with element will be written into an sortedArray;**

**(2) Do the above task until go over 2 parts;**

**\*/**

**do**

**{**

**while (l1 < r1 && array[l1] <= array[l2])**

**{**

**sortedArray[index] = array[l1];**

**l1++;**

**index++;**

**}**

**while (l2 < r2 && array[l2] <= array[l1])**

**{**

**sortedArray[index] = array[l2];**

**l2++;**

**index++;**

**}**

**}**

**while (l1 <r1 && l2<r2);**

**while (l1 < r1)**

**{**

**sortedArray[index] = array[l1];**

**index++;**

**l1++;**

**}**

**while (l2 < r2)**

**{**

**sortedArray[index] = array[l2];**

**index++;**

**l2++;**

**}**

**// Exit thread**

**pthread\_exit(NULL);**

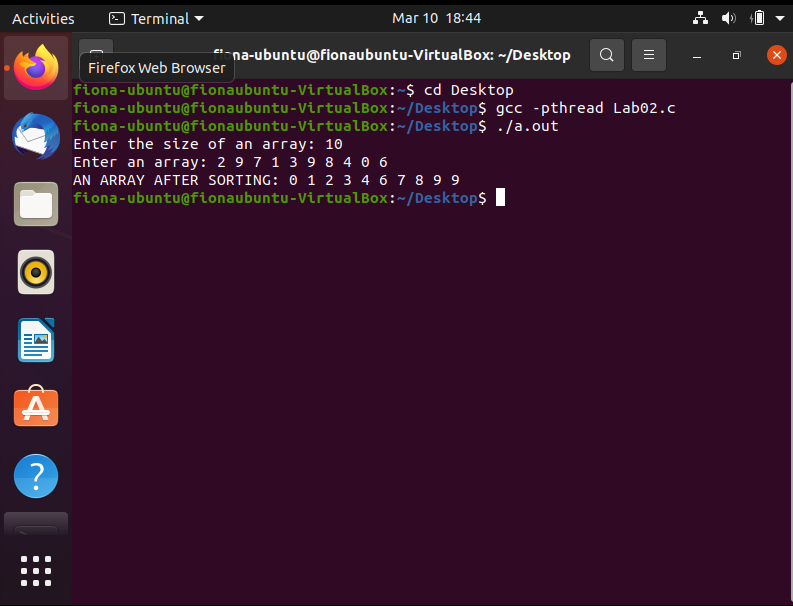
**}**

**II. Code Description:**

* **General part:**
* We use five global variables to share across all threads, which are:
  + A mutex object to lock shared data when the current thread accesses it.
  + Two arrays: one is the original array that the user inputs (unsortedArray), and another is the sorted array to store the array after sorting (sortedArray).
  + A variable called arraySize which is stores the size of an array
  + An ID of the current thread that is accessing shared data.
* In the main function, we create 3 threads. Two threads are used for sorting part (an array will be split into two parts, each thread will sort each part), and one thread is used for merging an array.
* After sorting and merging, we print the final result - the sorted array in the command line.
* **Sorting part:**
* We use a mutex to make sure that only one part of an array will be sorted at the time. If the ID of the current thread is 1, then the first part of an array (the element from 0 to arraySize/2 - 1) can be sorted; otherwise, the second part (element from arraySize/2 to arraySize - 1) will be sorted.
* We choose selection sort for the sorting function because it is quite simple and easy to fix.
* We do the sorting on the temporary array, and then we have to copy the data after sort into the provided array, which is an unsortedArray.
* Then at the end, we unlock the mutex, so the other thread can access the shared data and sort another part if the task does not complete.
* **Merging part:**
* Starting from the beginning of each part, we compare the element of each part. Then, we take the smaller element into the sortedArray, and increase the pointer of that part by 1 and continue comparing with the element which is pointed in another part.
* We do these tasks above until one part reaches its final element. If another part still has some elements that are not approached yet, we copy these elements into the last part of the sorted array.

**III. Results:**

The output is returned as a sorted number list.

****

**IV. Individual Contribution**

|  |  |
| --- | --- |
| **Task** | **Contributor** |
| Sorting part | Thanh Nguyen |
| Merging part | Fiona Le |
| Makefile | Thanh Nguyen |
| Report | Thanh Nguyen, Fiona Le |
| README | Fiona Le |
| Recording | Thanh Nguyen |