OS Project-3 Report

Multithreaded Sorting Application && Fork-Join Sorting Application

Zhou Yiyuan

516021910270
1025152261@qq.com

1 Multithreaded Sorting Application

1.1 Requirement

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term sorting threads) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread, a merging thread, which merges the two sublists into a single sorted list.

1.2 Code

- The input array to be sorted is defined as a global variable, which is available by all threads.
- A struct **range** is defined to pass parameters to threads.
- After creating the thread of sorting left part and the thread of sorting right part, **pthread_join()** is invoked, because merging must wait for complement of sorting left part and right part.
- The left part and the right part are sorted by Bubble Sort separately.

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

void* left_sorting(void*);
void* right_sorting(void*);
void* merge(void*);

void bubble_sorting(int *arr, int left, int right);

void Display(int *arr, int start, int end);

struct range{
    int start;
    int end;
};

int arr[10] = {5, 4, 10, 6, 9, 3, 8, 2, 1, 7};

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

    printf("Before sorting: ");
    Display(arr, 0, 9);

    pthread_t tid_left, tid_right, tid_merge;

    struct range range_left, range_right, range_merge;
```

```c
    range_left.start = 0;
    range_left.end = 4;

    range_right.start = 5;
    range_right.end = 9;

    range_merge.start = 0;
    range_merge.end = 9;

    //create threads to sort left part and right part
    pthread_create(&tid_left, NULL, left_sorting, (void*)&range_left);
    pthread_create(&tid_right, NULL, right_sorting, (void*)&range_right);
    //the thread of merging must wait threads of sorting left and right part
    pthread_join(tid_left, NULL);
    pthread_join(tid_right, NULL);
    //create a thread to merge
    pthread_create(&tid_merge, NULL, merge, (void*)&range_merge);
    pthread_join(tid_merge, NULL);

    printf("After sorting: ");
    Display(arr, 0, 9);

    return 0;
}

void Display(int *arr, int start, int end){
    for(int i=start; i<end+1; ++i){
        printf("%d ", arr[i]);
    }
    printf("\n");
}


void *left_sorting(void *arg){
    struct range *r;

    r = (struct range *)arg;

    printf("left part before sorting: ");
    Display(arr, 0, 4);

    bubble_sorting(arr, r->start, r->end);

    printf("left part after sorting: ");
    Display(arr, 0, 4);

    pthread_exit(NULL);
```

```
}

void *right_sorting(void *arg){
    struct range *r;
    r = (struct range *)arg;

    printf("right part before sorting: ");
    Display(arr, 5, 9);

    bubble_sorting(arr, r->start, r->end);

    printf("right part after sorting: ");
    Display(arr, 5, 9);

    pthread_exit(NULL);
}

void *merge(void *arg){
    struct range *r;
    r = (struct range *)arg;

    printf("before merging: ");
    Display(arr, 0, 9);

    int *tmp = (int *)malloc(10 * sizeof(int));

    int mid = (r->start + r->end) / 2;

    int i = 0;
    int j = mid + 1;
    int k = 0;

    while(i <= mid && j <= r->end){
        if(arr[i] <= arr[j]){
            tmp[k++] = arr[i++];
        }else{
            tmp[k++] = arr[j++];
        }
    }

    while(i <= mid){
        tmp[k++] = arr[i++];
    }

    while(j <= r->end + 1){
        tmp[k++] = arr[j++];
    }
    for(k=0; k<=9; k++){
        arr[k] = tmp[k];
    }

    printf("after merging: ");
    Display(arr, 0, 9);

    pthread_exit(NULL);

}

void bubble_sorting(int *arr, int left, int right){
    int i = 0;
    int j = 0;

    int flag = 0;

    for(i=0; i<=right-left; i++){
        flag = 0;
        for(j=left; j<=right-i-1; j++){
            if(arr[j] > arr[j+1]){
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
                flag = 1;
            }
        }
        if(flag == 0) break;
    }
}
```

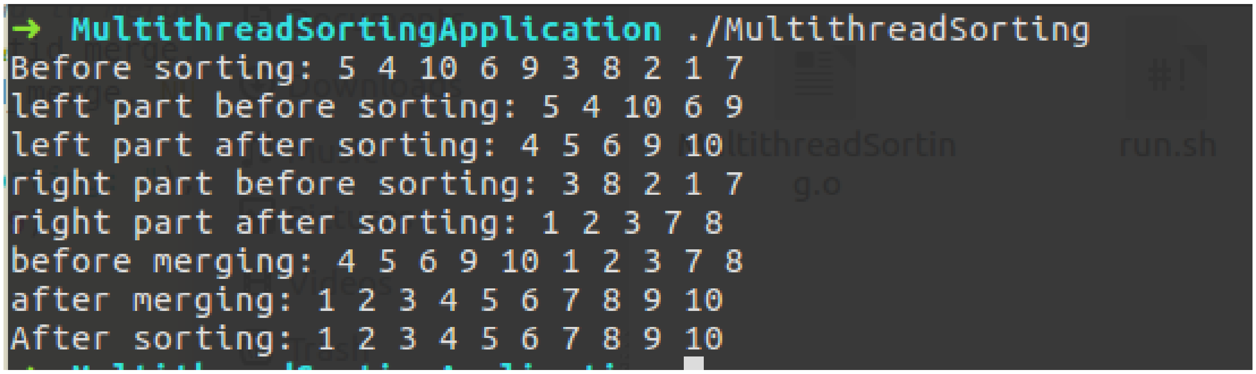
1.3 Result

The output of the program are shown as Figure 1.

2 Fork-Join Sorting Application

2.1 Requirement

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API . This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting algorithm:

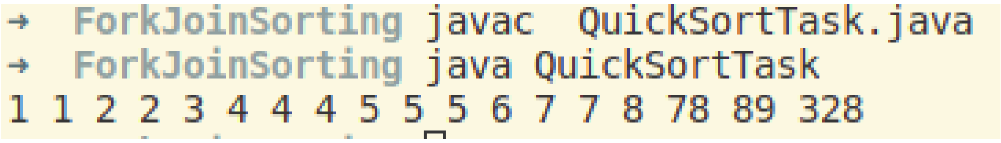


```
→ MultithreadSortingApplication ./MultithreadSorting
Before sorting: 5 4 10 6 9 3 8 2 1 7
left part before sorting: 5 4 10 6 9
left part after sorting: 4 5 6 9 10
right part before sorting: 3 8 2 1 7
right part after sorting: 1 2 3 7 8
before merging: 4 5 6 9 10 1 2 3 7 8
after merging: 1 2 3 4 5 6 7 8 9 10
After sorting: 1 2 3 4 5 6 7 8 9 10
```

Figure 1: The initial input is {5, 4, 10, 6, 9, 3, 8, 2, 1, 7}. Then two threads sort left part and right part separately and obtain {4, 5, 6, 9, 10, 1, 2, 3, 7, 8}. At last, a thread merge two parts and get the sorted array {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

3 QuickSort

3.1 Result



```
→  ForkJoinSorting javac  QuickSortTask.java
→  ForkJoinSorting java QuickSortTask
1 1 2 2 3 4 4 4 5 5 5 6 7 7 8 78 89 328
```

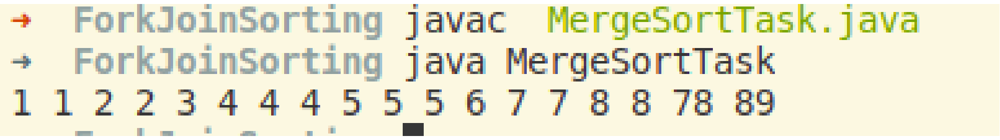
Figure 2: The result of QuickSort.

3.2 Code

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class QuickSortTask extends RecursiveAction{

    private int[] array;
    private int left;
    private int right;

    public QuickSortTask(int[] array, int left, int right) {
        this.array = array;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute() {
        int pivot = partition(array, left, right);
        QuickSortTask task1 = null;
        QuickSortTask task2 = null;
        if (pivot - left > 1) {
            task1 = new QuickSortTask(array, left, pivot-1);
            task1.fork();
        }
        if (right - pivot > 1) {
            task2 = new QuickSortTask(array, pivot+1, right);
            task2.fork();
        }
        if (task1 != null && !task1.isDone()) {
            task1.join();
        }
        if (task2 != null && !task2.isDone()) {
            task2.join();
        }
    }

    public static int partition(int[] a, int left, int right) {
        int pivot = a[left];
        while (left < right) {
            while (left < right && a[right] >= pivot) {
                right--;
            }
            swap(a, left, right);
            while (left < right && a[left] <= pivot) {
                left++;
            }
            swap(a, left, right);
        }
        return left;
    }

    public static void swap(int[] a, int i, int j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    public static void main(String[] args) {
        int[] a = {4,2,1,4,7,5,3,328,2,7,1,78,89,6,5,4,8,5};
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        QuickSortTask task = new QuickSortTask(a, 0, a.length-1);
        forkJoinPool.submit(task);
        try{
            Thread.sleep(1000);
        }catch(Exception e){
            System.out.println(e);
        }

        for (int n : a) {
            System.out.print(n + " ");
        }
        System.out.print("\n");
    }
}
```

4 MergeSort

4.1 Result



```
➜  ForkJoinSorting javac  MergeSortTask.java
➜  ForkJoinSorting java MergeSortTask
1 1 2 2 3 4 4 4 5 5 5 6 7 7 8 8 78 89
```

Figure 3: The result of MergeSort.

4.2 Code

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class MergeSortTask extends RecursiveAction{
    private int[] array;
    private int left;
    private int right;

    public static void main(String[] args){
        int[] a = {4,2,1,4,7,5,3,8,2,7,1,78,89,6,5,4,8,5};
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        MergeSortTask task = new MergeSortTask(a, 0, a.length-1);
        forkJoinPool.submit(task);
        try{
            Thread.sleep(1000);
        }catch(Exception e){
            System.out.println(e);
        }
        for (int n : a) {
            System.out.print(n + " ");
        }
        System.out.print("\n");
    }

    public MergeSortTask(int[] array, int left, int right){
        this.array = array;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute(){
        if(right - left > 0){
            int mid = (left + right) / 2;

            System.out.print("split:");
            for(int i=left; i<=right; ++i){
                System.out.print(array[i] + " ");
            }
            System.out.print(" ==> ");
            for(int i=left; i<=mid; ++i){
                System.out.print(array[i] + " ");
            }
            System.out.print(" || ");
            for(int i=mid+1; i<=right; ++i){
                System.out.print(array[i] + " ");
            }
            System.out.println("");


            MergeSortTask task1 = new MergeSortTask(array, left, mid);
            MergeSortTask task2 = new MergeSortTask(array, mid+1, right);
            invokeAll(task1, task2);
            // task1.fork();
            // task2.fork();
            task1.join();
            task2.join();
            merge(array, left, mid, right);
        }
    }

    public static void merge(int a[], int left, int mid, int right){
        int len = right - left + 1;
        int temp[] = new int[len];
        int i = left;
        int j = mid + 1;
        int k = 0;
        while(i<=mid && j<=right){
            if(a[i] <= a[j]){
                temp[k++] = a[i++];
            }else{
                temp[k++] = a[j++];
            }
        }
        while(i <= mid){
            temp[k++] = a[i++];
        }

        while(j <= right){
            temp[k++] = a[j++];
        }

        for(int s=0; s<temp.length; s++){
            a[left++] = temp[s];
        }
    }
}
```