EI338 Computer Systems Engineering

Project 3

November 29, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise 1 Multithreaded Sorting Application

In this exercise, we are asked to sort a series of integers using multithreaded method. The basic idea is very simple. We first divide the list into two sublists, each of which is then sorted by an individual thread. Then in the main thread, after waiting for these two sub-threads, do the merge.

Basic Struture

The basic structure includes initializing the data, assigning threads with tasks, and merge.

```
int main() {
    FILE *fp = fopen("test_2.txt", "r");

    parameters sort_data[2], merge_data;
    pthread_t sort_tids[2], merge_tid;
    pthread_attr_t sort_attrs[2], merge_attr;

    int m = 5;

    /*load the sudoku girds */
        for (int i = 0; i < 10; ++i) {
                fscanf(fp, "%d", &data[i]);
        }

    /* initialize data */
    sort_data[0].low = 0;
    sort_data[0].high = m;
    sort_data[1].low = m;
    sort_data[1].high = 10;
    merge_data.low = 0;
    merge_data.high = 10;

    /* set the default attributes of the threads */
    pthread_attr_init(&sort_attrs[0]);
    pthread_attr_init(&sort_attrs[1]);
    pthread_attr_init(&merge_attr);

    /* create the thread */
    pthread_create(&sort_tids[0], &sort_attrs[0], sorting, &sort_data[0]);
```

```
    pthread_create(&sort_tids[1], &sort_attrs[1], sorting, &sort_data[1]);

    /* wait for the thread to exit */
    pthread_join(sort_tids[0], NULL);
    pthread_join(sort_tids[1], NULL);

    /* merge */
    pthread_create(&merge_tid, &merge_attr, merging, &merge_data);
    pthread_join(merge_tid, NULL);

    /* output */
    for (int i = 0; i < 10; ++i) {
        printf("%d ", sorted_data[i]);
    }

    printf("\n");

    return 0;
}
```

Listing 1: main()

Sort

sort() will serve as the function to sort two sublists.

```
/* sort data */
void *sorting(void *param) {
    parameters *param_tmp = (parameters*) param;
    int min_index, tmp;

    for (int i = param_tmp->low; i < param_tmp->high - 1; ++i) {
                min_index = i;

        for(int j = i + 1; j < param_tmp->high; ++j) {
                        if(data[min_index] > data[j]) {
                                min_index = j;
                        }
                }

                tmp = data[i];
                data[i] = data[min_index];
                data[min_index] = tmp;
        }
}
```

Listing 2: Sort

Merge

After two sublists are sorted separately, we will call another thread to merge sublists to get the final result.

```
/* merge data */
void *merging(void *param) {
    parameters *param_tmp = (parameters*) param;
```

```
    int m = (param_tmp->low + param_tmp->high) / 2;
    int i = param_tmp->low;
    int j = m;
    int k = i;

    while (i < m || j < param_tmp->high)
    {
        if (i == m) {
            sorted_data[k++] = data[j++];
            continue;
        }

        if (j == param_tmp->high) {
            sorted_data[k++] = data[i++];
            continue;
        }

        if (data[i] < data[j]) {
            sorted_data[k++] = data[i++];
        }
        else {
            sorted_data[k++] = data[j++];
        }
    }
}
```

Listing 3: Merge

Result

A simple test shows our currectness.

Figure 1: test 1

Exercise 2 Fork-Join Sorting Application

This time, we revise the implementation in Exercise 1 using Java's fork-join parallelism API. Two versions will be implemented:

1. Quicksort
2. Mergesort

QuickSortTask

The Quicksort implementation will use the Quicksort algorithm for dividing the list of elements to be sorted into a left half and a right half based on the position of the pivot value.

```
public class QuickSortTask<T extends Comparable<T>> extends SortTask<T> {
    public QuickSortTask(int begin, int end, T[] array) {
        super(begin, end, array);
    }
```

```
    @Override
    // special sort: quick sort
    protected void Sort() {
        // find pivot
        int m = Pivot();

        // quick sort two sub-arrays
        QuickSortTask<T> leftTask = new QuickSortTask<T>(begin, m, array);
        QuickSortTask<T> rightTask = new QuickSortTask<T>(m + 1, end, array);

        leftTask.fork();
        rightTask.fork();

        leftTask.join();
        rightTask.join();
    }

    // find pivot and adjust
    private int Pivot() {
        // some code
    }
}
```

Listing 4: QuickSortTask

MergeSortTask

The Mergesort algorithm will divide the list into two evenly sized halves.

```
public class MergeSortTask<T extends Comparable<T>> extends SortTask<T> {

    private int middle;

    public MergeSortTask(int begin, int end, T[] array) {
        super(begin, end, array);
        middle = (begin + end) / 2;
    }

    @Override
    // special sort: merge sort
    protected void Sort() {
        // merge sort two sub-arrays
        MergeSortTask<T> leftTask = new MergeSortTask<T>(begin, middle, array);
        MergeSortTask<T> rightTask = new MergeSortTask<T>(middle, end, array);

        leftTask.fork();
        rightTask.fork();

        leftTask.join();
        rightTask.join();

        // merge
        Merge();
    }
```

```
    // merge sorted array
    private void Merge() {
        // some code
    }
}
```

Listing 5: MergeSortTask

SortTask

The class **SumTask** extends RecursiveTask.

```
public abstract class SortTask<T extends Comparable<T>> extends RecursiveAction {
    private static final int THRESHOLD = 100;

    protected int begin;
    protected int end;
    protected T[] array;

    public SortTask(int begin, int end, T[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    @Override
    protected void compute() {
        Sort();
    }

    protected abstract void Sort();
}
```

Listing 6: SortTask