CSE4034 Project 2 Report

(Kevser İLDEŞ – 150116048)

The task was implementing a parallel merge of sorted integers stored in N input files using concurent threads. A thread can read from two file sor two shared buffers and write to a final output file or another shared buffer. Therefore, firstly I defined buffers, as 2d arrays, and semaphores globally and defined a struct for threads to specify all features.

Here is the struct and its variables:

```
typedef struct{
    int index;
    int isFromFile;
    int isToFile;
    char file1[15];
    char file2[15];
    char outFile[15];
    int bufindex_1;
    int bufindex_2;
    int bufwriteindex;
}threadReadWrite;
```

Inside this struct, i have index of thread, isTromFile and isToFile integer values to check and file name values accordingly and if from or to buffer, i have indices for buffer arrays.

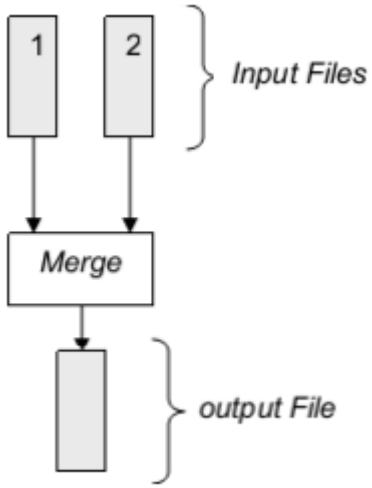
Then, I implemented a function to initialize all semaphores with initial value 0. In main function, at first I checked number of arguments and provide actual usage if it is not given correctly. Then, I called init_sems function and assigned number of buffers and threads to be used according to number of files. Next, I defined two arrays: on a for thread ids which is threadTable and other is for structs those will be given as argument to threads. After that, in a loop, checking number of files and indices, i assigned struct values of each thread accordingly and then created thread with this struct. At the end, after the loop, there is another loop to wait for all threads to join.

In the merge function threads will execute, firstly, I implemented a check condition with number of files and index of thread to make necessary threads wait till previous ones completed their jobs using sem_wait. Then, i made assignments of struct variables in local to use easily inside the function. If thread would read from file, I opened files with given names for reading with fopen and took number of integer values of each then took first numbers in files, otherwise if it reads from buffer, I calculated number of elements in buffer and again took first numbers. Next, if it would write to a file, I opened a file to write with given names with fopen.

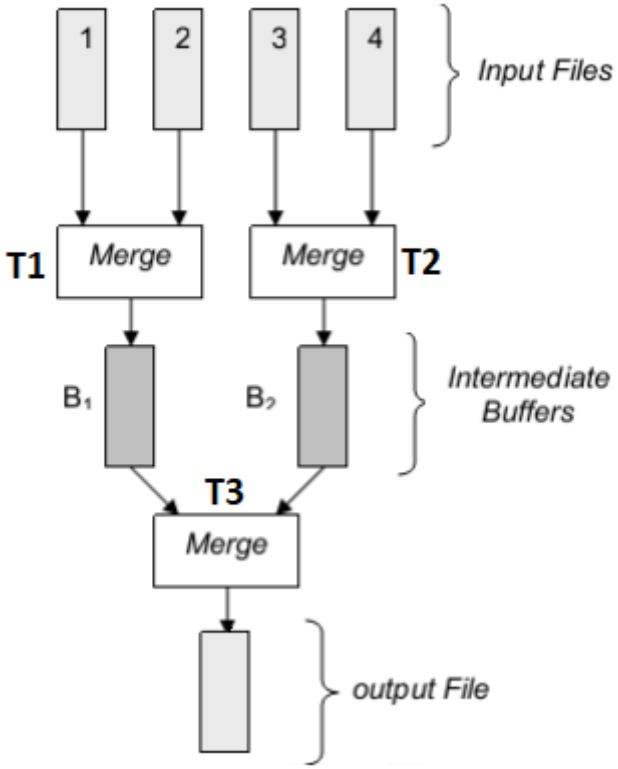
Then, I implemented merge operation in a for loop. Firstly, it compares two numbers and smaller one is written to file or buffer according to conditions checked and next number is taken from file or buffer again with if statements using assigned values of struct. Then after main loop, it checks whether there are numbers left in one of them, it writes them to end with same logic.

Finally, after a thread finishes its job, it controls whether to post a semaphore or not using again number of files and index value and let other thread waiting for it to go on.

Before moving onto the execution examples, below, logic for threads to wait or post is provided:

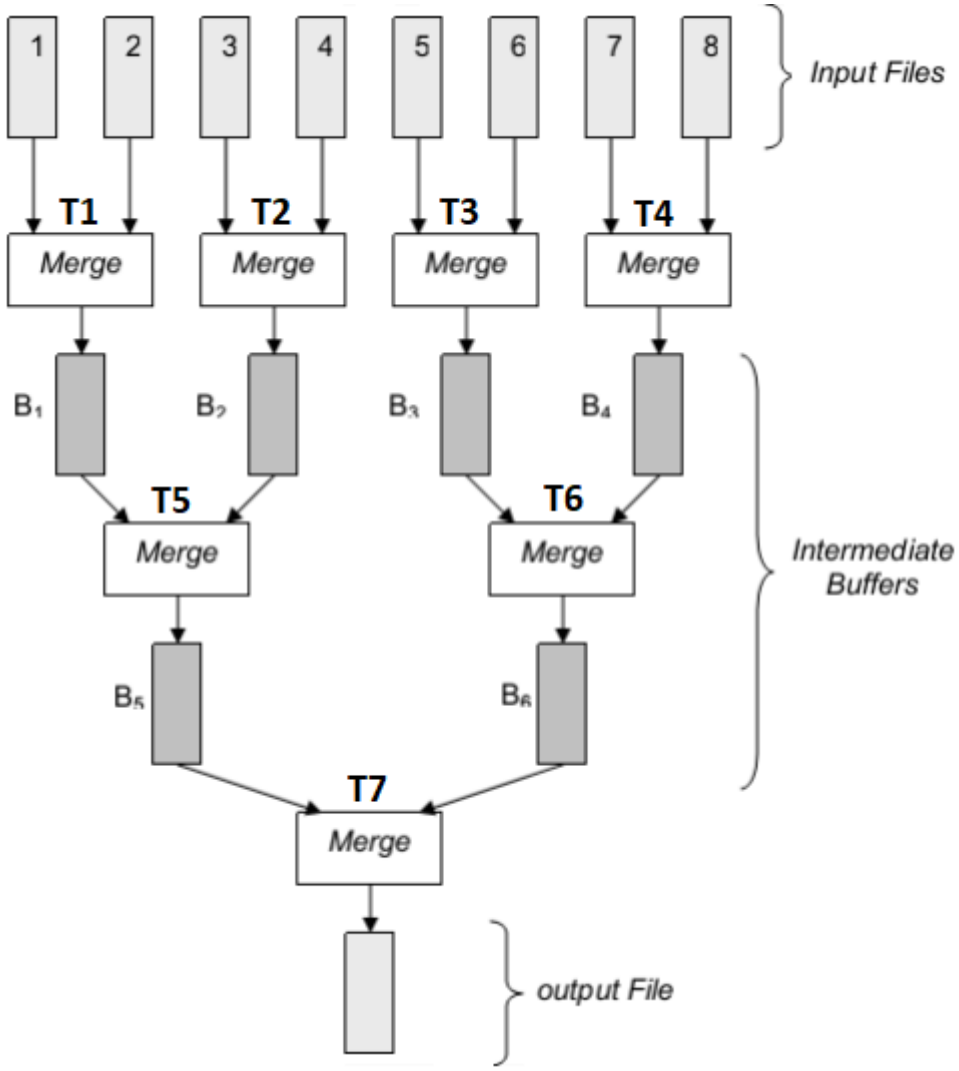


For two files, there are only one thread and thus no semaphores used.



For four files, there are two buffers and three threads used and in this case, as thread 3 uses the buffers those would be filled by thread one and two, it should wait them to finish. Therefore, fort his case, one semaphore is enough to control, which will be posted by T1 and T2 at the end of their execution for once and T3 will wait in the beginning of its run twice.

And for eight files, there would be seven threads and six buffers. Thread one to four can run without waiting. But as Thread five and six uses buffers those would be filled by them, T5 should wait for T1 and T2 to finish their execution and T6 should wait T3 and T4. As so, T7 should wait for T5 and T6. For al of these, a total of three semaphores would be enough. First one will be posted at the end of T1 and T3 once and waited in the beginning of T5 twice; second one will be posted by T3 and T4 at the end once and waited in the beginning of T6 twice; and finally last one will be posted at the end of T5 and T6 once and waited by T7 in the beginning of the execution.



And here are example runs of the code:

For 2 files
Execution:

```
kevserildes@kevserildes-Lenovo-IdeaPad-Z500:~/Desktop/unix/p2$ gcc arraydeneme.c -o arraydeneme.o -lpthread
kevserildes@kevserildes-Lenovo-IdeaPad-Z500:~/Desktop/unix/p2$ ./arraydeneme.o -n 2 in1.txt in2.txt -o out.txt
Thread 0
Thread 0 goes
Thread 0 ends...
```

Files:

out.txt

```
3
5
7
8
9
9
9
10
14
17
19
22
25
27
30
```

in1.txt

```
8
3
5
9
9
14
17
25
30
```

in2.txt

```
7
7
8
9
10
19
22
27
```

For 4 files:

Execution:

```
kevserildes@kevserildes-Lenovo-IdeaPad-Z500:~/Desktop/unix/p2$ ./arraydeneme.o -n 4 i1.txt i2.txt i3.txt i4.txt -o out4.txt
Thread 0
Thread 1
Thread 0 goes
Thread 2
Thread 1 goes
Thread 2 waits
Thread 0 ends...
Thread 1 ends...
Thread 2 goes
Thread 2 ends...
```

Files:

out4.txt

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

i1.txt

```
5
1
8
9
16
17
```

i2.txt

```
8
2
7
10
15
18
23
24
26
```

i3.txt

```
6
3
6
11
14
19
22
```

i4.txt

```
7
4
5
12
13
20
21
25
```

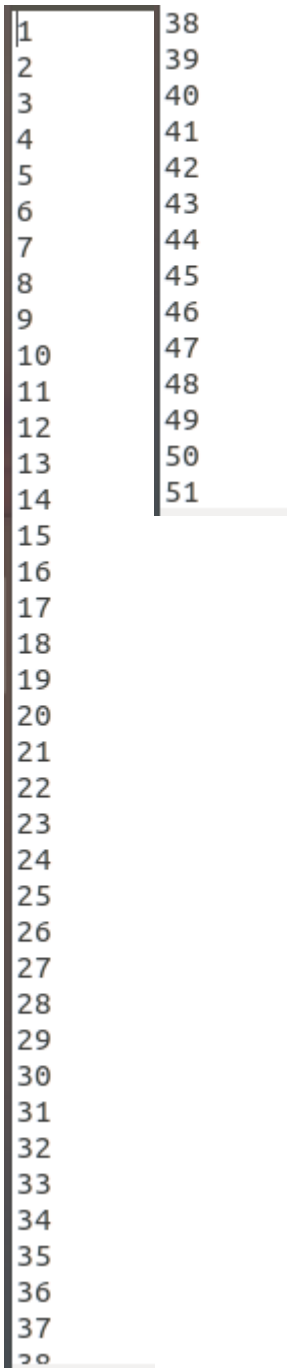
For 8 files

Execution:

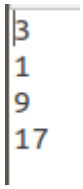
```
kevserildes@kevserildes-Lenovo-IdeaPad-Z500:~/Desktop/unix/p2$ ./arraydeneme.o -n 8 d1.txt d2.txt d3.txt d4.txt d5.txt d6.txt d7.txt d8.txt
-o out8.txt
Thread 0
Thread 1
Thread 2
Thread 1 goes
Thread 0 goes
Thread 2 goes
Thread 3
Thread 1 ends...
Thread 2 ends...
Thread 4
Thread 3 goes
Thread 5
Thread 4 waits
Thread 3 ends...
Thread 6
Thread 5 waits
Thread 5 goes
Thread 5 ends...
Thread 0 ends...
Thread 6 waits
Thread 4 goes
Thread 4 ends...
Thread 6 goes
Thread 6 ends...
```

Files:

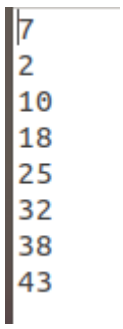
out8.txt



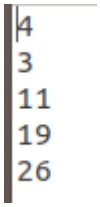
d1.txt



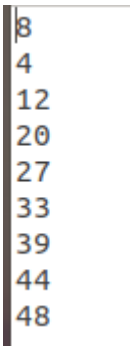
d2.txt



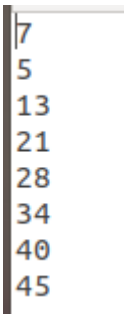
d3.txt



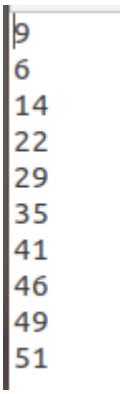
d4.txt



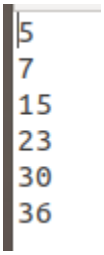
d5.txt



d6.txt



d7.txt



d8.txt