Radix Sort Float Point Numbers with Memory Map

Implement a C program that sorts a set of 4-byte float point values in ***ascending order*** using radix sort. ***The values are saved in a file***. The program should read/write the file through memory mapping. When the program finishes, the sorted values should be saved in the same file.

Submission Instructions

1. Submit: 1) your C program, and 2) one or two screenshots in jpg format showing that your program really works.
2. Name your scripts using the pattern NJITID#.c, and name your screenshots using the pattern NJITID#_index.jpg. NJITID# is the eight-digit NJIT ID (Not your UCID, Rutgers students also have NJIT IDs). Index is the index number showing the order of the screenshots (e.g., 1 or 2).
3. Submit individual files. DO NOT SUBMIT A ZIP FILE.

Objectives

- To gain more experience with using bitwise operations
- To understand the binary format of float point numbers
- To gain more understanding on radix-sort
- To learn how to read/write files using memory mapping

Requirements and Instructions

- Your program should take one argument, which is the pathname of the file containing the data to be sorted. For example, to sort the float point values saved in *./file5k*, you can use the following command:

  ```
  ./radixsort   ./file5k
  ```

- The number of float-point values saved in the file can be calculated using file size and the size of each float point value (i.e., 4 bytes). Thus, there is no need to specify the number of values.
- To access the data in the file, your program needs to use memory mapping. Avoid using conventional calls, e.g., read(), fread(), write(), or fwrite(), to read/write the file. Refer to the slides on *Linux File and Directory Operations*, particularly the two examples in the *Memory-mapped files* part, for how to read/write files using memory mapping.
- Your program must use radix-sort and work directly on the **binary** data (i.e., avoid converting the data into strings or characters). To use binary operations to extract bits from a float-point number, you need to use a union to include two types, float and int. For example, the following code is to extract the least significant bit from float point number 0.1.

```
unsigned int b;
union ufi {
    float f;
    int i;
} u;
u.f = 0.1;
b = u.i & 0x1;
```

- Since the program sorts the numbers using binary format, your program only needs **two buckets** to help with sorting. The memory space that maps the file can be used to merge the data. This reduces memory space consumption. At the same time, when the last round of radix-sort, the file automatically saves the sorted values.
- Some of the float point values are negative. Special attention is needed to handle the problem caused by sign bits. Refer to the slides on *Radix Sort* for the methods handling signed float point numbers.
- You can compile `gendata.c` attached with this assignment and use it to generate random values and save them into a file. The executable file can also be found in /bin in the virtual machine. The program also reports the sum of the values. For example, to generate 5000 random values and save them into ./file5kvalues, you can use the following command
  `./gendata 5000 ./file5kvalues`
- You can compile `checkdata.c` attached with this assignment and use it to check whether the float point values have been sorted in ascending order. The executable file can also be found in /bin in the virtual machine. The tool also calculates a sum of the values in the file. Thus, you can compare the sum with the sum reported by `gendata`. The two sums should be very similar with minor numerical error caused by limited precisions.
  `./checkdata ./file5kvalues`
- Optimize your implementation. For example, to copy a large number of numbers, you can use `memcpy` instead of copying the numbers one by one.

Testing

Test 1. The program can correctly sort 1 million float point values in a file within 1 minute, and the file can pass the test with the *checkdata* program (i.e., sorted; the sum of sorted values is close to the sum of unsorted values given by *gendata* when the file was created (difference <5%))

Step 1: Generate a file containing 1 million float point values using `gendata`, and write down the sum of these values reported by `gendata`:

```
gendata 1000000 ./file1mvalues
```

Step 2: run the program to sort the values:

```
time ./major_hw3_1 ./file1mvalues
```

Step 3: check whether the values have been sorted using `checkdata`, and write down the sum reported by `checkdata`:

```
checkdata ./ file1mvalues
```

Step 4: compare the sum reported by `gendata` and the sum reported by `checkdata`.

Test 2. The program can correctly sort 100 million float point values in a file within 2 minute, and the file can pass the test with the *checkdata* program (i.e., sorted; the sum of sorted values is close to the sum of unsorted values given by *gendata* when the file was created (difference <5%))

Step 1: Generate a file containing 100 million float point values using `gendata`, and write down the sum of these values reported by `gendata`:

```
gendata 100000000 ./file100mvalues
```

Step 2: run the program to sort the values:

```
time ./major_hw3_1 ./file100mvalues
```

Step 3: check whether the values have been sorted using *checkdata*, and write down the sum reported by *checkdata*:

```
checkdata ./ file100mvalues
```

Step 4: compare the sum report by *gendata* and the sum reported by *checkdata*.