Problem 1: Sort Floating Point Numbers Saved in a File

Implement **TWO** C programs. Each program sorts a set of 4-byte floating point values in ***ascending order***. The values are saved in a file. When the program finishes, the sorted values should be saved in the same file.
One program should read/write the file using system calls (e.g., `read()` or `write()`) or C library functions (e.g., `fread()` and `fwrite()`). The other program should read/write the file through memory mapping.

Submission Instructions
Submit individual files following the instructions below. DO NOT SUBMIT A ZIP FILE.

- **Your C programs**: Name your programs *problem1_normal.c* and *problem1_mmap.c*.
- **1~2 screenshots in jpg format for each program showing that your program really works**: Name your screenshots using the pattern problem1_normal_index.jpg or problem1_mmap_index.jpg . Index is 1, 2, or, 3…
- **1 screenshot in jpg format to show the difference between the two programs**. You may use *diff* to get the differences and then take a screenshot. Name your screenshot problem1_diff.jpg

```
diff   problem1_normal.c   problem1_mmap.c
```

Objectives

- To learn how to access the data in files.
- To learn how to read/write files using memory mapping.
- To learn how to test a program.

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 floating point values saved in *./file5k*, you can use the following command:

```
./your_program   ./file5k
```

- You can choose the sorting algorithm you prefer. But it must be reasonably fast, because we may test your programs with a large amount of data (check the testing section below).
- The number of floating-point values saved in the file can be calculated using file size and the size of each floating point value (i.e., 4 bytes). Thus, there is no need to specify the number of values.
- To access the data in the file, one program should use system calls (e.g., `read()` or `write()`) or C library functions (e.g., `fread()` and `fwrite()`). The other program should use memory mapping. (Avoid using system calls or library functions in this program.) 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.
- 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 floating 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 floating 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 floating 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 ./your_program ./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 floating 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 floating 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 ./your_program ./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`.

Problem 2: Traverse a Directory

Write **TWO** C programs. The programs print out the pathnames of all the files in a directory, including all the files under all the sub-directories, sub-sub-directories, etc. (i.e., sub-directories at all the levels). The pathnames should be printed out in order of file sizes with the pathname of the smallest file on top (i.e., ascending). One use recursion and one does not use recursion.

Submission Instructions

Submit individual files following the instructions below. DO NOT SUBMIT A ZIP FILE.

- **Your C programs**: Name your programs *problem2_resursive.c* and *problem2_nonresursive.c*.
- **1~2 screenshots in jpg format for each program showing that your program really works**: Name your screenshots using the pattern problem2_recursive_index.jpg or problem2_nonrecursive_index.jpg . Index is 1, 2, 3…
- **1 screenshot in jpg format to show the difference between the two programs**. On the screenshot, **highlight or circle the code that does the recursion**. You may use `diff` to get the differences and then take a screenshot. Name your screenshot problem2_diff.jpg

```
diff   problem2_resursive.c   problem2_nonresursive.c
```

Objectives

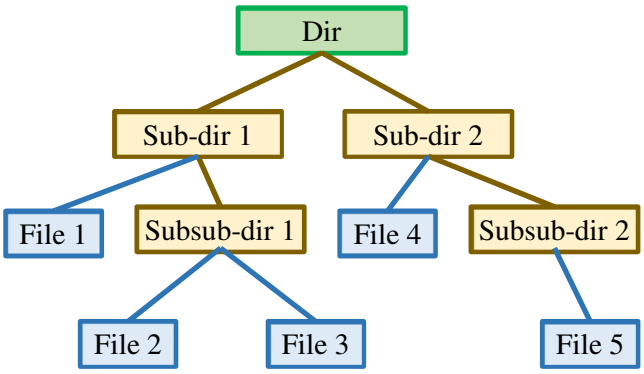
- To learn how to traverse a directory
- To learn how to check file status
- To gain more experience on using pointers and linked lists
- To gain more experience on sorting a linked list.
- To learn how to write recursive functions
- To learn how to test a program.

Requirements and Instructions

- Your program should take one argument, which is the pathname of the directory. (Important! If you don't follow strictly, we may not be able to run your program correctly, and you may get lower grades.) For example, the following command sorts the files under directory `/usr/share/man`:

```
./your_program  /usr/share/man
```

- All files directly and indirectly under the directory should be considered. This means that your program needs to traverse the sub-directories, sub-sub-directories, sub-sub-sub-directories, etc, and check the files there. For example, when your program is run against directory `Dir` on the right, all 5 files (`File 1` ~ `File 5`) should be included.



- For an implementation using recursion, refer to the *monitor* example in the slides for how to traverse a directory tree in a C program, particularly the code in *MonitorFile* and *processDirectory* functions. For an implementation without using recursion, use a linked list to organize the sub-directories that have not been scanned. Refer to the slides on directory traversal in the bash part. The slides on state space search can also give you some ideas.
- To simplify the problem, the program only needs to consider regular files (i.e., those with the `S_ISREG()` test returning 1). Also, in the problem, we assume that each regular file only has one hard link. So, the program does not need to look at the number of hard links when checking files.
- Symbolic links are links, and are not considered as regular files. Thus, when your program checks the status of a file, make sure that it calls `lstat()`, not `stat()`. Check the slides for the differences between `lstat()` and `stat()`.
- Assume the longest pathname does not exceed 256 single-byte characters.
- For the files with the same size, there is no requirement on how their pathnames should be sorted.
- Print out the results on the screen in text format, one line for each file with file size followed by a tabular symbol ('\t') and the file pathname. Programs using different formats in outputs will fail the tests and lead to lower grades.
  The sample output when the program is run with the command below against directory /usr/share/man/ is shown as follows:

```
$./your_program  /usr/share/man
30      /usr/share/man/man1/cpp-4.8.1.gz
30      /usr/share/man/man1/cpp-5.1.gz
30      /usr/share/man/man1/gcc-4.8.1.gz
30      /usr/share/man/man1/gcc-5.1.gz
35      /usr/share/man/man3/XauDisposeAuth.3.gz
35      /usr/share/man/man3/XauFileName.3.gz
35      /usr/share/man/man3/XauGetAuthByAddr.3.gz
…
86618   /usr/share/man/man1/bash.1.gz
104755  /usr/share/man/man5/smb.conf.5.gz
287622  /usr/share/man/man1/x86_64-linux-gnu-gcc-6.1.gz
303306  /usr/share/man/man1/x86_64-linux-gnu-g++-7.1.gz
303306  /usr/share/man/man1/x86_64-linux-gnu-gcc-7.1.gz
```

Testing: For each program, perform the following tests:

- *Test 1:* The program can print out correct result when it is run against a directory containing a few regular files with different sizes

  Step 1. Create a directory and a few files in the directory

```
mkdir ./test1
dd if=/dev/zero of=./test1/file1 bs=10 count=1
dd if=/dev/zero of=./test1/file2 bs=100 count=1
```

```
dd if=/dev/zero of=./test1/file3 bs=1000 count=1
```

Step 2. Execute the program against `./test1`:

```
./your_program   ./test1
```

Step 3. Check the output of the program, which should look like

```
10      ./test1/file1
100     ./test1/file2
1000    ./test1/file3
```

- *Test 2:* The program can print out correct result when it is run against a directory containing sub-directories and regular files in sub-directories

Step 1. Create a directory and a few files in the directory

```
mkdir ./test2
dd if=/dev/zero of=./test2/file1 bs=10 count=1
dd if=/dev/zero of=./test2/file2 bs=100 count=1
mkdir ./test2/dir1
dd if=/dev/zero of=./test2/dir1/file3 bs=20 count=1
dd if=/dev/zero of=./test2/dir1/file4 bs=200 count=1
mkdir ./test2/dir2
dd if=/dev/zero of=./test2/dir2/file5 bs=30 count=1
dd if=/dev/zero of=./test2/dir2/file6 bs=300 count=1
mkdir ./test2/dir3
dd if=/dev/zero of=./test2/dir3/file7 bs=40 count=1
dd if=/dev/zero of=./test2/dir3/file8 bs=400 count=1
```

Step 2. Execute the program against `./test2`:

```
./your_program   ./test2
```

Step 3. Check the output of the program, which should look like

```
10      ./test1/file1
20      ./test1/dir1/file3
30      ./test1/dir2/file5
40      ./test1/dir3/file7
100     ./test1/file2
200     ./test1/dir1/file4
300     ./test1/dir2/file6
400     ./test1/dir3/file8
```

- *Test 3:* The program can print out correct result when it is run against directory /usr/share/man.

Step 1. Run the program and redirect the result into a file *output_test.txt*

```
./your_program /usr/share/man > output_test.txt
```

Step 2. Check whether the sizes are correctly sorted in the output. If they are correctly sorted, you will not see *diff* print out any text.

```
cut -f 1 output_test.txt > sizes_test.txt
sort -s -n sizes_test.txt > sizes_test_sorted.txt
diff sizes_test.txt sizes_test_sorted.txt
```

Step 3. Run the following command on one line and redirect the result into another file *output_standard.txt*

```
find /usr/share/man -type f -print0 | xargs -0r du -bl | sort
  -k 1,1n -k 2,2 > output_standard.txt
```

Step 4. This step checks whether the output includes the sizes of any files. Compare the sizes contained in *output_test.txt* and *output_standard.txt* using the following commands. You should not see *diff* print out any text if the sizes of all the files have been correctly collected in the program.

```
cut -f 1 output_standard.txt > sizes_standard.txt
diff sizes_test_sorted.txt sizes_standard.txt
```

Step 5. This step tests whether the output includes the pathnames of all the files. Compare the pathnames in *output_test.txt* and *output_standard.txt* using the following commands. You should not see diff print out any text if the pathnames of all the files have been correctly collected in the program.

```
cut -f 2 output_test.txt | sort > filelist_test.txt
cut -f 2 output_standard.txt | sort > filelist_standard.txt
diff filelist_test.txt filelist_standard.txt
```