NC STATE UNIVERSITY

CSC 541
Assignment 3
Disk-Based Mergesort

Introduction

The goals of this assignment are two-fold:

1. To introduce you to sorting data on disk using mergesort.
2. To compare the performance of different algorithms for creating and merging runs during mergesort.

Index File

During this assignment you will sort a binary index file of integer key values. The values are stored in the file in a random order. You will use a mergesort to produce a second index file whose key values are sorted in ascending order.

Program Execution

Your program will be named `assn_3` and it will run from the command line. Three command line arguments will be specified: a mergesort method, the name of the input index file, and the name of the sorted output index file.

```
assn_3 mergesort-method index-file sorted-index-file
```

Your program must support three different mergesort methods.

1.	`--basic`	Split the index file into sorted runs stored on disk, then merge the runs to produce a sorted index file.
2.	`--multistep`	Split the index file into sorted runs. Merge subsets of runs to create super-runs, then merge the super-runs to produce a sorted index file.
3.	`--replacement`	Split the index file into sorted runs created using replacement selection, then merge the runs to produce a sorted index file.

For example, executing your program as follows

```
assn_3 --multistep input.bin sort.bin
```

would apply a multistep mergesort to `input.bin` to sort it ascending by key value. The result would be stored in `sort.bin`.

Note. For convenience, we refer to the input index file as `input.bin` and the output sorted index file as `sort.bin` throughout the remainder of the assignment.

Available Memory

Mergesort's run sizes and merge performance depend on the amount of memory available for run creating and merging runs.

Your program will be assigned one input buffer for reading data (*e.g.*, blocks of keys during run creation of parts of runs during merging). The input buffer must be sized to hold a maximum of 1000 integer keys.

Your program will also be assigned one output buffer for writing data (*e.g.*, sorted blocks of keys during run creation or sorted subsets of `sort.bin` during merging). The output buffer must be sized to hold a maximum of 1000 integer keys.

Basic Mergesort

If your program sees the merge method `--basic`, it will implement a standard mergesort of the keys in `input.bin`. The program should perform the following steps.

1. Open `input.bin` and read its contents in 1000-key blocks using the input buffer.
2. Sort each block and write it to disk as a run file. You can use any in-memory sorting algorithm you want (e.g., C's `qsort()` function). Name each run file `index-file.n`, where `n` is a 3-digit run identifier, starting at 0. For example, if the input index file is `input.bin`, the run files would be named

```
input.bin.000
input.bin.001
input.bin.002
 ...
```

3. Open each run file and buffer part of its contents into the input buffer. The amount of each run you can buffer will depend on how many runs you are merging (*e.g.*, merging 50 runs using the 1000-key input buffer allows you to buffer 20 keys per run).
4. Merge the runs to produce sorted output. Use the output buffer to write the results in 1000-key chunks as binary data to `sort.bin`.
5. Whenever a run's buffer is exhausted, read another block from the run file. Continue until all run files are exhausted.

You must record how much time it takes to complete the basic mergesort. This includes run creation, merging, and writing the results to `sort.bin`.

Note. You will never be asked to merge more than 1000 runs in Step 3. This guarantees there will always be enough memory to assign a non-empty buffer to each run.

Multistep Mergesort

If your program sees the merge method `--multistep`, it will implement a two-step mergesort of the keys in `input.bin`. The program should perform the following steps.

1. Create the initial runs for `input.bin`, exactly like the basic mergesort.
2. Merge a set of 15 runs to produce a super-run. Open the first 15 run files and buffer them using your input buffer. Merge the 15 runs to produce sorted output, using your output buffer to write the results as binary data to a super-run file.
3. Continue merging sets of 15 runs until all of the runs have been processed. Name each super-run file `index-file.super.n`, where `n` is a 3-digit super-run identifier, starting at 0. For example, if the input file is `input.bin`, the super-run files would be named

```
input.bin.super.000
input.bin.super.001
input.bin.super.002
 ...
```

   Note. If the number of runs created in Step 1 is not evenly divisible by 15, the final super-run will merge fewer than 15 runs.

4. Merge all of the super-runs to produce sorted output. Use the input buffer to read part of the contents of each super-run. Use the output buffer to write the results in 1000-key chunks as binary data to `sort.bin`.

You must record how much time it takes to complete the multistep mergesort. This includes initial run creation, merging to create super-runs, merging super-runs, and writing the results to `sort.bin`.

Note. You will never be asked to merge more than 1000 super-runs in Step 3. This guarantees there will always be enough memory to assign a non-empty buffer to each super-run.

Replacement Selection Mergesort

If your program sees the merge method `--replacement`, it will implement a mergesort that uses replacement selection to create runs from the values in `input.bin`. The program should perform the following steps.

1. Divide your input buffer into two parts: 750 entries are reserved for a heap H_1 ... H_{750}, and 250 entries are reserved as an input buffer B_1 ... B_{250} to read keys from `input.bin`.
2. Read the first 750 keys from `input.bin` into H, and the next 250 keys into B. Rearrange H so it forms an ascending heap.
3. Append H_1 (the smallest value in the heap) to the current run, managed through the output buffer. Use replacement selection to determine where to place B_1.
   - If $H_1 \leq B_1$, replace H_1 with B_1.
   - If $H_1 > B_1$, replace H_1 with H_{750}, reducing the size of the heap by one. Replace H_{750} with B_1, increasing the size of the secondary heap by one.

   Adjust H_1 to reform H into a heap.
4. Continue replacement selection until H is empty, at which point the current run is completed. The secondary heap will be full, so it replaces H, and a new run is started.
5. Run creation continues until all values in `input.bin` have been processed. Name the runs exactly as you did for the basic mergesort (*i.e.*, `input.bin.000`, `input.bin.001`, and so on).
6. Merge the runs to produce sorted output, exactly like the merge step in the basic mergesort.

You must record how much time it takes to complete the replacement selection mergesort. This includes replacement selection run creation, merging the replacement selection runs, and writing the results to `sort.bin`.

Note. You will never be asked to merge more than 1000 runs in Step 6. This guarantees there will always be enough memory to assign a non-empty buffer to each run.

Programming Environment

All programs must be written in C, and compiled to run on the `remote.eos.ncsu.edu` Linux server. Any ssh client can be used to access your Unity account and AFS storage space on this machine.

Measuring Time

The simplest way to measure execution time is to use `gettimeofday()` to query the current time at appropriate locations in your program.

```
#include <sys/time.h>

struct timeval tm;

gettimeofday( &tm, NULL );
printf( "Seconds: %d\n", tm.tv_sec );
printf( "Microseconds: %d\n", tm.tv_usec );
```

Comparing `tv_sec` and `tv_usec` for two `timeval` structs will allow you to measure the amount of time that's passed between two `gettimeofday()` calls.

Writing Results

Sorted keys must be written to `sort.bin` as binary integers. C's built-in file writing operations allow this to be done very easily.

```
#include <stdio.h>

FILE *fp;                  /* Output file stream */
int   output_buf[ 1000 ];  /* Output buffer */

fp = fopen( "sort.bin", "wb" );
fwrite( output_buf, sizeof( int ), 1000, fp );
fclose( fp );
```

Your program must also print the total execution time for the mergesort it performs as a single line on-screen. Assuming the execution time is held in a `timeval` struct called `exec_tm`, use the following `printf` statement to do this.

```
printf( "Time: %ld.%06ld", exec_tm.tv_sec, exec_tm.tv_usec );
```

Your assignment will be run automatically, and the output it produces will be compared to known, correct output using diff. Because of this, **your output must conform to the above requirements exactly**. If it doesn't, diff will report your output as incorrect, and it will be marked accordingly.

Supplemental Material

In order to help you test your program, we provide example input.bin and sort.bin files.

- input.bin, a binary input file file containing 25,000 keys,
- sort.bin, a binary output file containing input.bin's 25,000 keys in ascending sorted order.

You can use diff to compare output from your program to our sort.bin file.

If you want to compare your intermediate output with what we expect, you can download and uncompress the following files that are produced during the basic, multistep, and replacement selection mergesort. Each file is zip'd to save space.

- basic mergesort
- multistep mergesort
- replacement selecton mergesort

Please remember, the files we're providing here are meant to serve as examples only. Apart from holding integers, and a guarantee that the number of runs (or super-runs) will not exceed the input buffer's capacity, **you cannot make any assumptions** about the size or the content of the input and sort files we will use to test your program.

diff will only tell you if two binary files are the same or different. It will not tell you where they differ. If you want to investigate exactly *why* your binary files differ from ours, you can download and compile this program.

- dump.c

Running dump binfile will dump the contents of a packed binary integer file as text, one line per integer. You can redirect results from dump to generate text versions of the two binary files you want to compare, then diff those text files.

```
% gcc -o dump dump.c
% dump input.bin.000 > input.txt.000
% dump my-input.bin.000 > my-input.txt.000
% diff input.txt.000 my-input.txt.000
```

Now, diff will tell you exactly which lines (*i.e.,* which integers) are different.

Hand-In Requirements

Use Moodle (the online assignment submission software) to submit the following files:

- assn_3, a Linux executable of your finished assignment, and
- all associated source code files (these can be called anything you want).

There are four important requirements that your assignment must satisfy.

1. Your executable file must be named exactly as shown above. The program will be run and marked electronically using a script file, so using a different name means the executable will not be found, and subsequently will not be marked.
2. Your program must be compiled to run on remote.eos.ncsu.edu. If we cannot run your program, we will not be able to mark it, and we will be forced to assign you a grade of 0.
3. Your program must produce output that exactly matches the format described in the Writing Results section of this assignment. If it doesn't, it will not pass our automatic comparison to known, correct output.
4. You must submit your source code with your executable prior to the submission deadline. If you do not submit your source code, we cannot MOSS it to check for code similarity. Because of this, any assignment that does not include source code will be assigned a grade of 0.

Updated 29-Nov-19