COMS W3157 Advanced Programming, Lab #4

Please refer to our instructions for lab retrieval and submission. You must include the required information in your README.txt. This lab consists of multiple parts, so your code for each part must be in a subdirectory (named "part1", "part2", etc.). Like before, you are required to have 5 Git commits with meaningful commit messages, a Makefile for each part where you write code, and Valgrind output in your README.

Part 1: Message database search program

(a)

The directory ~j-hui/cs3157-pub/bin/ contains a simple database into which you can put records, each consisting of a name and a short message. Here are the files in that directory:

- mdb-cs3157: This is the database file, a binary file that contains each record one after another. Each record is 40 bytes, so the size of this database file is always a multiple of 40.

- mdb-add-cs3157: This program inserts a record into the mdb-cs3157 database. It will prompt you for a name and a short message, which it uses to fill in the following structure:

  ```
  struct MdbRec {
      char name[16];
      char  msg[24];
  };
  ```

  The structure in memory is then written to the end of the database file. Note that the name and message will be truncated to 15 and 23 characters (not including null byte), respectively, to fit them into the structure.

- mdb-lookup-cs3157: This program searches for a particular name or message in the database file. It will prompt you for a string to search for. If you simply press ENTER, it will show you all the records in the database. If you type something, it will show you the records containing what you typed, either in the name field or in the msg field.

  Only the first 5 letters are used in the search. So searching for "hello" and "helloooooo" will yield the same result. The match is case-sensitive.

  The program keeps running, prompting you for another string to search for. You can press Ctrl-D instead of entering any input to terminate the program.

Part 1(a) is easy. You play with mdb-add-cs3157 and mdb-lookup-cs3157, inserting a couple of records and seeing the entertaining messages that your classmates have put in.

List the name and message pairs you have inserted into the database in your README.txt file. You are required to insert at least one record into the database.

(b)

There are two more programs in ~j-hui/cs3157-pub/bin/. mdb-add and mdb-lookup are similar to mdb-add-cs3157 and mdb-lookup-cs3157, but they let you work with your own database file. For example, you can insert records into your own database file by running:

```
    mdb-add my-mdb
```

It will create a database file named my-mdb in your current directory if the file is not there already.

Your job is to write mdb-lookup. It should behave the same way as my mdb-lookup. You are not required to write mdb-add (it won't be graded, but you are encouraged to try!).

Here are some hints and programming requirements:

- Put the MdbRec structure definition in a header file called mdb.h.

- Name your executable "mdb-lookup".

- mdb-lookup takes the database file name as its sole command line argument.

- When the program starts, you must first read all records from the database file into memory. Moreover, the records MUST be kept in a linked list using the mylist library from Lab 3.

- You do not need to know the size of the database file. Instead, you must read the database one record at a time, until you have read all the records. Do NOT read the entire file into memory at once.

- Just like in part 2 of Lab 3, you must not bring any of the mylist library files into your lab4 directory; instead, use the -I, -L, and -l flags. You MUST use the header file and library archive we have installed in /home/j-hui/cs3157-pub/include and /home/j-hui/cs3157-pub/lib, respectively. Do NOT use your own solutions.

- You must keep the records in the linked list in the same order that they were in the database file. addFront() function is not a good choice since it has the effect of inverting the order as you insert the records. Use addAfter() instead. See mylist-test.c from Lab 3 for an example of using addAfter() to append items at the end of a linked list.

- Our Lab 3 solutions may contain additional functions that you were not required to implement, such as addBack(). Do NOT use addBack() for Lab 4.

- mdb-lookup should truncate the search string at 5 characters, not including the newline character.

- When reading from the keyboard, you must read in the entire line first, and then take only the first 5 characters of it. A common mistake is to read only 5 characters from the keyboard. Then, the input in the next iteration will begin at the 6th character of the previous line, which is not correct. You may not assume a maximum input line length. In other words, you need to handle reading an arbitrary number of characters until you encounter a newline; only search using up to the first 5.

- Some useful functions for taking user input are:

  ```
  strncpy(), strlen(), fgets()
  ```

  Note that strncpy() may or may not null-terminate the destination string, and fgets() may or may not include newline character in the buffer. Read their man pages carefully, and if in doubt, test their behavior yourself.

If you are writing mdb-add (optional), you will also find isprint() useful.

- Don’t forget to print those records that contain the given search string in any of the two fields. The library function strstr() may come in handy.

- Don’t forget to print the record number when you print out the records, in exactly the same way that my implementation does, i.e., the record numbers are the positions of the records in the database file, starting from 1. For record numbers with 3 digits or less, we pad the output to 4 characters using the %4d format specifier.

- Remember that at the prompt, pressing Ctrl-D without any other input terminates mdb-lookup. You should design your loop to detect this condition, so that you can get out of the loop and clean up. In particular, don’t forget to close all files you opened and free() all heap memory.

- Don’t forget to check for memory leaks and errors using Valgrind; include your Valgrind output in your README.txt.

Part 2: Files with holes & measuring the effect of I/O buffering (optional)

Part 2 of this lab is strictly optional and will not be graded. You don’t have to do it if you don’t have time, and you will not be expected to know this material in future coursework.

You do not need to use Valgrind for Part 2.

(a)

Write a program called "hole" (your C file should be hole.c) that creates a file with a large hole. Here is how you do it:

- Open a file named "file-with-hole" for writing.

- Write a single character ’A’ into beginning of the file.

- Go to the 1,000,000th byte position of the file using fseek().

- Write a single character ’Z’ into the file.

- Close the file and exit.

You will have created the file-with-hole file, and its size will be 1000001 bytes (verify this with "ls -l").

Your job is to determine whether that file actually takes up 1 MB of disk space. Study the man pages of the following commands, look for relevant command line options, and use them to investigate the actual disk usage of file-with-hole:

- ls
- od
- du

Report your finding in README.txt. Support your findings by including the output of those commands and commenting on the relevant parts of that output.

(b)

Write a simple program "copy" (copy.c) that reads from stdin and writes to stdout character by character using getchar() and putchar(). This program is actually in K&R2, p17, which you are allowed to use.

Use this program to make a copy of file-with-hole and call it "file-copied". Here is how:

```
    ./copy < file-with-hole > file-copied
```

How much disk space is the new file using? Perform the same investigation as in part (a) using ls, od, du. Report your findings in README.txt.

Would you say the contents of those files are same or different? In what sense are they same or different? What does the command "cmp" tell you? Discuss these questions in README.txt.

(c)

Make another program, "copy-nobuf" (copy-nobuf.c), that is identical to "copy" except that the buffering on stdin and stdout are turned off, using setbuf().

Perform the copying from "file-with-hole" to "file-copied" using "copy" and "copy-nobuf", and compare the running times using the "time" command.

Which one is faster, by how much? Why do you think that is? Discuss your measurements in README.txt.

--

Make sure you perform all Part 2 experiments on the same machine (i.e., CLAC). Results may vary between machines, depending on the hardware and operating system.

--

Good luck!