COMS W3157 Advanced Programming, Lab #4

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

Please refer to the lab retrieval and submission instruction.

If this lab consists of multiple parts, your code for each part must be in a subdirectory, named "part1", "part2", etc.

If a part asks for a program, you must provide a Makefile. Please refer to lab1 for the requirements for Makefiles. The TAs will make no attempt to compile your program other than typing "make".

Please include a README.txt in the top level directory.

Checking memory errors with valgrind

You will be heavily penalized if your program contains memory errors. Memory errors include (among other things) failure to call free() on the memory you obtained through malloc(), accessing past array bounds, dereferencing uninitialized pointers, etc.

You can use a debugging tool called "valgrind" to check your code:

```
valgrind --leak-check=yes ./your_executable
```

It will tell you if your program has any memory error. See "The Valgrind Quick Start Guide" at http://valgrind.org/docs/manual/quick-start.html for more info.

You must include the output of the valgrind run for EACH PART in your README.txt, unless you are told explicitly that you don't have to. In addition, TAs will run valgrind on your program when grading.

Part 1: Message database search program

(a)

The directory "/home/jae/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. It's 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 is a program that inserts a record into the mdb-cs3157 database file. It will ask for a name and a short message, and then fills up the following structure with them:

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

The structure in memory is then written at the end of the database file. Note that the name and the message will be truncated to 15 and 23 characters, respectively, in order to fit them into the structure.

mdb-lookup-cs3157

- This is the program you use to see what's in the database and search for a particular name or a message. It will prompt 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 those records that contain 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 "hellooooo" 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 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.

(b)

There are two more programs in /home/jae/cs3157-pub/bin. mdb-add and mdb-lookup are similar to the 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 program that behaves 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!

As usual, here are some hints and programming requirements:

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

- Name your executables "mdb-lookup".

- mdb-lookup take the database file name as the 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 because you must read the database one record at a time in a loop until you have read all the records. Do NOT read the entire file at once into memory.

- Just like in part 2 of lab 3, you must not bring any of the mylist library files into lab4 directory. Use -I and -L flags to use the header file and the library file directly from the "../../lab3/solutions/part1" directory. Note the use of a relative path. Do NOT use an absolute path or a path that include your UNI. They will not be accessible when the TAs try to build your code so your build will fail if you use them. Also, make sure you use the solution version of mylist library, not your own version.

- 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 the example of using addAfter() to append items at the end of a linked list.

- Lab 3 solution may contain some additional functions that were not required such as addBack(). Do NOT use addBack() for lab 4.

- When reading from the keyboard, you must read in the entire line first, and then take only the first n characters of it. A common mistake would be to read only n characters from the keyboard. Then, the input in the next iteration will begin at the (n+1)th character of the previous line, which is not what you want.

- You may assume that an input line will be well under 1000 characters -- i.e., call fgets() with 'char line[1000]'. For an input line longer than that, it's ok to treat it as multiple input lines -- i.e., you don't need to do anything special as long as your code doesn't hang, crash, or produce memory errors.

- mdb-lookup should truncate the search string at 5 characters.

- Some useful functions for taking user input are:

  strncpy(), strlen(), fgets()

  Note that strncpy() may or may not null-terminate the string, and fgets() may or may not include newline character in the buffer. Read their descriptions in the book or man page very carefully.

  If you are writing mdb-add for your own fun and learning, 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 the same way that my version does; i.e., the record numbers are the positions of the records in the database file, starting from 1.

- Remember that you terminate mdb-lookup by pressing Ctrl-D. Ctrl-D generates the EOF on the standard input. You should design your loop to detect it, so that you can get out of the loop and clean up. In particular, don't forget to free all memories you allocated and close all files you opened.

- Don't forget to check for memory error using valgrind, and include valgrind output in your README.txt.

Part 2: Files with holes & Measuring the effect of I/O buffering (0 points)

Part 2 of this lab is optional and will not be graded. You don't have to do it if you don't have time. If you have time, do it for learning. Do it for fun!

(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 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, find out any command line options that might help in this case, and use them to investigate the question of actual disk usage:

```
ls, od, du
```

Report your finding in README.txt. Support your finding by including the outputs of the commands and giving interpretations of the relevant parts of the outputs.

(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 you do it:

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

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

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 the measurement results in your README.txt.

--

Make sure you perform all experiments in Part 2 on one of the clic machines. Results on other operating systems may be different. You do not need to use valgrind for Part 2.

--

Good luck!