CS 352 (Fall 16): System Programming and UNIX

Project #7

Party in the NSA

due at 9pm, Wed 2 Nov 2016

1 Overview

It's time for some data mining! In this project, your program will open up a set of files (usually more than one), each of which is full of pairs of phone numbers (representing phone calls). You will record how many times each phone number has talked to each other phone number. At the end, you will be given a pair of numbers: you will search through the data to find out if the two numbers have called each other (or had a less-direct connection).

In this project, you'll exercise several new techniques:

- Using command-line arguments (and testcases that will use them)
- Using `fopen()` and `fclose()`
- Using `free()`

Within the C language, you will be exploring lists of lists - that is, a data structure where each node of a linked list contains another list, with more information.

1.1 New Requirements

Your program must, in addition to previous requirements, also:

- free() **every** malloc()'d buffer before the testcase terminates (even if it terminates with a serious error).
- Close all files (we'll test this by doing some crazy testcases which use lots and lots of files).
- New, updated testcase format (detailed below).

2 Program - Phone Calls

In this project, your program will take several arguments on the command line. These arguments will include one or more file names; the last two arguments will be phone numbers.

Your program will read from the files listed (it will **not** read from `stdin`). You will open each one in turn, and read its contents; you will store all of the information that you find into a data structure in memory. You will mix together all of the information from all of the files, into a single data structure.

After you have read all of the files, you will look at the two phone numbers (the last two arguments on the command line). You will then search through your data structures and find out how many times (if at all) these two numbers have had a phone call. You will report this number.

Next, you will look for any case where the two numbers have both contacted a third person (that is, they have a mutual acquaintance). If there is any such third person, you will also report that. (For testing purposes - so we don't have to sort the output - you will only indicate whether a third person exists. You will not print out what number it is.)

2.1 Input Format

Each line of each input file will contain two seven-digit phone numbers, separated by a single space. Each phone number must be of the form `xxx-xxxx`; there must be no extra leading or trailing whitespace (or any other characters) on any line. The two numbers given on the line must be different; a phone cannot call itself!

If any line does not exactly follow this format, then you should report it as an error, and ignored.

Any completely empty line (with nothing on it except a newline) should be ignored silently (it is not an error).

If any file is empty (that is, it contains nothing, or only blank lines), you should report an error, but keep running the program.

2.2 Command-Line Argument Format

Your program will be run as follows:

```
<progName> <file1> <file2> ... <phone1> <phone2>
```

The phone numbers will be given in the same 7-digit format as the input files. If either number does not follow this format, then report an error and terminate the program with no other output.

The command line must give at least 1 file name to open (but could be arbitrarily long). If it does not have at least one file name (or if it doesn't have enough arguments to even have phone numbers), then report an error and terminate the program with no other output.

If any file cannot be opened, print out an error, but **keep running.**

2.3 Output Format

Your program (unless it terminates early because of a serious error) should always print out exactly two lines of output. The first should contain a single integer; it is the number of times that the two phone numbers have had a conversation.

The second line will be either `yes` or `no`. Print out `yes` if the two numbers have any connection in common; print out `no` if not.

2.4 Example Output

For example, consider the following input file (for simplicity, we'll assume that all of the input is in a single file):

```
123-4567 123-9999
123-4567 123-0000
123-4567 123-2020
123-9999 123-4567
123-0000 123-2020
123-9999 123-4567
123-9999 123-0000
```

Now, assume that the two numbers are `123-4567` and `123-9999`. The proper output for this file would be:

```
3
yes
```

It is 3 because there were 3 calls between them (notice how you have to pay attention to calls that happen in both directions). It is `yes` because both phone numbers had connections to `123-0000`.

Suppose that the input was:

```
000-0000 777-7777
999-9999 777-7777
```

If the two numbers were `000-0000` and `999-9999`, the proper output would be:

```
0
yes
```

Finally, suppose that the input was:

```
111-1111 222-2222
222-2222 333-3333
333-3333 444-4444
444-4444 555-5555
111-1111 555-5555
```

If the two numbers were `111-1111` and `555-5555`, the proper output would be:

```
1
no
```

2.5 Example Testcase

I have provided a single example testcase in the project directory. You will find that it has four files:

```
test_phoneCalls_01_exampleTestcase/
    CMDLINE
    foo
    bar
    baz
```

Look at the contents of each file; foo, bar, and baz are input files for your program, and CMDLINE gives the command-line arguments which should be passed to the program. The contents of CMDLINE are:

```
foo bar baz 746-6322 557-7765
```

So this testcase will pass all three files to the program, and then search for the two numbers given.

2.6 Error Conditions

The error conditions are listed above. As always, the program should exit with status 0 if there were no errors, and 1 if there were one or more errors.

As always, check for bad return values from `malloc()` and such; if an error occurs, print out an error message and exit with status 1 immediately.

3 Data Structure

There are a variety of ways that you could implement the data structures for this project. However, we want you to practice a certain technique - and for that reason, we're going to apply some rules.

Your program must store the information about the calls as a linked-list of linked-lists. That is, you will have a linked list, and each node represents a single phone number. Each of these **nodes** has another list **inside** it - and the nodes on this second list represent phone numbers which have conversed with the first number.

This will require that you declare two different structs. The first struct, which represents a node on the first list, should have a data field (to store the phone number), a next field (to go to the next element on the list), and a field to store the 2nd list. (I like to use head pointers - but any sort of way of "storing a list" is OK here.)

The second struct, which represents one node in the smaller list, will have a next field as well - but it will **only** point to other nodes in the same short list. In addition, this node should contain another phone number, and a counter, indicating how many times the two phone numbers have been in contact.

3.1 Sorting Not Required

You are not required to sort either the large, outer list or the smaller lists. They **may** be out of order. However, I encourage you to keep them sorted - as your

program may be a tiny bit faster, and it certainly will be easier to look at the data structures in `gdb`!

3.2 Symmetric or Not?

When phone number X calls Y, there are two places that you might insert this information into the data structure:

- Search for X on the main list; then add Y to the little list inside X.
- Search for Y on the main list; then add X to the little list inside Y.

Either strategy is legal in this program - but remember that if you insert in two different ways, then each search you do (like, asking how many phone calls happened between them) will need to happen **twice.**

This is my recommendation as an **old, experienced programmer:** insert the data in **both** places. Sure, it's twice as much memory - but making your program readazble is often even more important.

4 Grading

As with Project 6, we will not provide a grading script, or any testcases. You must provide testcases which cover your entire code (except for `malloc()` error handling), and your code must run without `valgrind` errors. In addition, you must provide a Makefile which builds your code. See the Project 6 spec for a reference.

4.1 New Testcase Format

In this Project, you will use a **new format for testcases;** this is because we now need to support command-line arguments. In this program, each testcase will contain a special file, called `CMDLINE`, which is the command line for the testcase:

```
phoneCalls/
    Makefile
    phoneCalls.c
    test_phoneCalls_01_simpleList/
        CMDLINE
        file1
        file2
        file3
```

Make sure that any filename that you give in the CMDLINE file actually exists - and it has to be part of this testcase. (Unless, of course, you are testing to see what happens when you give it a filename that we cannot.)

See the project directory online for one example testcase.

4.2 Grading Scheme

This project has some grade items which apply to the entire project (not to individual testcases), so our grading scheme has gotten more complex:

- 10% of your grade will come from your Makefile (see requirements below).
- 20% of your grade will come from `gcov` coverage (see requirements below).
- 70% of your grade will come from testing your code, comparing it to the example program.

Of course, just as before, we also have a set of penalties which may be applied - these subtract from your overall score:

- -10% - You use any data structure other than "list of lists"
- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)
- -10% - Missing checks of returns codes from standard library functions (such as, but not limited to, `malloc()`)
- -20% - Any use of `scanf()` family `%s` specifier without limiting the number of characters read.

4.3 Testcase Grading

We have decided to subdivide the score from each testcase:

- You must exactly match `stdout` to get **ANY** credit for a testcase.
- You will lose one-quarter of the credit for the testcase if the exit status or `stderr` do not match the example executable.
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors.

As always, you lose half of your testcase points if your code does not compile cleanly (that is, without warnings using `-Wall`).

4.4 Testcase Selection

As in Project 6, we will use the testcases that you submit to perform `gcov` testing on your program - and a selection of testcases from the students (plus a few from the instructors) for checking to see if your program works **correctly.**

5 `valgrind` - NEW Requirements

In Project 6, we required that you run with no valgrind errors. In this project, we'll add one more requirement: you must free **all** of your malloc()'d buffers before the program ends.

This is easy to test. Simply add the option

```
--leak-check=full
```

to the valgrind command-line. When you do that, valgrind will report any not-yet-freed memory as an error (at the end of your program).

6 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`
- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

7 `turnin`

Turn in the following directory structure using the assignment name `cs352_f16_proj07`:

```
phoneCalls/
    Makefile
    phoneCalls.c
    test_pohneCalls_*/
        <various files in various testcase directories>     (turn in several of these!)
```

REMEMBER: Submit only this one directory! Do not place it inside another directory!

8 Grade Preview

48 hours before the project is due, we will run the automated grading script on whatever code has been turned in at that time; we'll email you the result. This will give you a chance to see if there are any issues that you've overlooked so far, which will cost you points.

Of course, this grading script will not include the full set of student testcases, which we will use in the grading at the end, but it will give you a reasonable idea of what sort of score you might earn later on.

We will only do this once for this project. If you want to take advantage of this opportunity, then make sure that you have working code (which compiles!) turned in by 48 hours before the due date.