MAT 4370: Programming Assignment 10

Due: Friday, March 22

Checking for Memory Leaks

As discussed in class, having a memory leak in a C program is a serious bug — yet it can be challenging to find these leaks. Fortunately, there are software tools which help us locate leaks. For the first part of this assignment, you will install one such tool and then use it on a program known to have a leak.

Installing Valgrind

The tool we will use is known as Valgrind, described at its website, `http://valgrind.org/`. Here is a brief summary of how to install this tool in the Macintosh lab. (Installation for other Unix-like systems is similar.)

1. Download the source code for the current release of the software, a file named:

   ```
   valgrind-3.8.1.tar.bz2
   ```

   The suffix `.bz2` indicates it is a compressed file and `.tar` indicates it is an archive. In the Unix world, both of these are quite common.

2. Using the Finder, move this file to your home directory.

3. Again with the Finder, uncompress and extract the files in the archive by double clicking on its icon. This will produce a folder named `valgrind-3.8.1`. (The compressed version you downloaded can now be discarded.)

4. In the `valgrind-3.8.1` directory there is a file named `README`. (Open that file and read it.) Of particular importance is the section "to install from a tar.bz2 distribution." Don't do what it says just yet — these steps are described below.

5. Open a terminal window and change directories to `valgrind-3.8.1`.

6. Determine your home directory:

   ```
   echo ~
   ```

   In response, this should give you something like: `/external/Users/jcollege`, assuming your name is Joe College.

7. Configure Valgrind, by giving a command similar to the following:

   ```
   ./configure --prefix=/external/Users/jcollege/valgrind
   ```

   (Use your home directory, not Joe's!)

8. In response, you will receive a number of confirmations and see evidence of a number of actions.

9. Compile the Valgrind software suite:

   ```
   make
   ```

   In response, the `gcc` compiler will be used to compile a number of C source files, which will take a few minutes. You will see hundreds of lines of text scroll past your terminal window.

10. Install the Valgrind software suite:

    ```
    make install
    ```

    If all went well, you will now have an installation of Valgrind stored in a folder named `valgrind`. Using the Finder, go look for this now. When you find it, descend along the following path:

    ```
    valgrind/share/doc/valgrind
    ```

    There you will find extensive documentation for this software.

11. Now that you have the Valgrind software, you need to adjust your `PATH` so the system can find it. To do this:

    (a) Edit the file named `.profile`, found in your home directory. If you don't already have such a file, then create it. As the final line in this file, add the following:

    ```
    PATH=$PATH:~/bin:~/valgrind/bin
    ```

    You are asking for two directories to be added to your `PATH`: one for your own personal binaries, and one for those of Valgrind.

    (b) After editing this file, open a new terminal window and check the value of `PATH`:

    ```
    echo $PATH
    ```

    In response, you should see a sequence of paths, ending with the directory where the Valgrind programs are stored.

    (c) Check it out. Try this:

    ```
    valgrind ls -l
    ```

    You should get a summary which shows 0 leaks and 0 errors.

Using Valgrind

As a first application of Valgrind, do the following:

1. Get a copy of `incorrect-readfile5.c` from the course website, found in the Homework 9 code examples. Put it and a copy of your `words` file in some convenient place. Compile (with the `-g` debugging switch, saving the executable output in `irf`) and run this program. It should produce correct output, although we know that it has two places that cause memory leaks.

2. Use Valgrind to check the execution. To use Valgrind, give the following commmand:

    ```
    valgrind ./irf
    ```

    For more informative output, use this command instead, carefully observing the results:

    ```
    valgrind --leak-check=full ./irf
    ```

3. Redirect the results of Valgrind to a file named `valgrind-results`:

    ```
    valgrind --leak-check=full ./irf 2> valgrind-results
    ```

    You will submit this file as evidence that you have successfully installed and executed Valgrind.

4. Add Valgrind to your future workflow. In addition to your own careful implementation and testing effort, you should now run all of your C programs with Valgrind to help detect any memory leaks.

Extending Past Work

We want to remove some of the deficiencies of the "poor man's sort" program you recently implemented. To do this, use the following outline:

1. Download a copy of the source files for this assignment. When you expand this archive, you will find a previous solution to the problem of reading lines from a text file — organized in a more modular fashion. Examine the contents of each file.

2. In addition to the modularization, there is now a `Makefile` which streamlines the compilation process. To build the program, use the following command:

   ```
   make readfile
   ```

   In response to this command, notice that `gcc` is used first to compile `util.c` and `readfile.c` and then again, but this time it links the object files `util.o` and `readfile.o` to create the executable file `readfile`.

3. Run the compiled program to see if it behaves as you expect. After this, run it again with Valgrind to check for memory leaks.

4. We are using `readfile.c` as a stepping stone to the poor man's sorting program. Notice that `readfile.c` does not store more than one line at a time. That's fine for this application, but ultimately we want to be able to store all of the lines. We have done this before, but the lines were stored in a fixed-size array. We can do better.

   To improve upon this, implement a function with the following prototype:

   ```
   char **read_lines(FILE *fp);
   ```

   The idea is that this function should read all lines in a given file and store them in a NULL-terminated array of strings. The array should grow in an efficient way (by doubling) and upon return should be adjusted to the correct size. Put the prototype in `util.h`, put its implementation in `util.c`, then modify `main()` so it uses this function.

5. Implement three additional utility functions, with the following prototypes:

   ```
   int count_lines(char *p[ ]);
   void display_lines(char *p[ ]);
   void display_lines_in_reverse(char *p[ ]);
   ```

   The first of these returns the number of strings stored in a given array; the other two display the strings on the standard output file — either from the first to last, or the other way around. As before, put the prototypes in `util.h` file and the implementations in `util.c`.

6. To prove to yourself that these functions are working correctly, modify `readfile.c` so it displays the contents of the file twice: once in each direction. (Of course, it now goes without saying that you will use Valgrind to test for memory leaks.)

7. When you have convinced yourself that `read_lines()` function is working, you can move on to the poor man's sort revision. Revise code as necessary in order to simplify the code, making use of the utility functions. At this point in time, continue to use selection sort for the sorting process: as before, it is the file I/O and string manipulation that is most important. Add a `pmsort` target to the `Makefile` to allow for compilation and linking to occur. Test your work.

8. If we are serious about systems programming, we need to realize that other people will be using our programs, so efficiency matters. To get a sense of just how bad selection sort is for large files, let's time how long it takes to sort a file with approximately 200,000 lines.

   To measure the time it takes your program, use the following command:

   ```
   time ./pmsort words200K
   ```

   Be prepared to wait! I won't spoil the surprise and tell you how long it takes, but this might be a good time to relax or think about the next step. When your program finishes, it is the "user" time that is of most interest. Record this time in a file named `my-time1`.

9. You may recall from a course on data structures that there are many known sorting algorithms. The C standard library includes a particularly efficient algorithm, `qsort()`, an optimized version of the famous Quicksort algorithm. Modify your sort program to make use of this function. A discussion of `qsort()` begins on page 440 of our textbook. How much time does it take your revised sort program to process the system dictionary file? Record the user time in a file named `my-time2`.

What to Submit

When you have completed all the assigned work, put the following files in a folder named `hw10` and submit it in the usual way. You only need to submit the most up-to-date sorting program — I don't need to see the version which uses selection sort.

```
Makefile
util.h
util.c
readfile.c
pmsort.c
my-time1
my-time2
valgrind-results
```