CS 352 (Summer 2021): Systems Programming and UNIX

Project #5.2 (50% grade of complete project 5)

`malloc()` and `free()` with strings; `valgrind`; Makefiles

due at 11:30pm, Thursday 8 Jul 2021

1 Overview

This project has only one program. At a first glance, it doesn't seem particularly remarkable - but when I wrote the solutions, I found that it took some care in order to get it right (in all of the corner cases). So start early, and debug carefully!

In this project, we'll be doing something similar to the `intRollingBuffer` program - except that this time, the buffer will not be allocated with `malloc()`. Instead, it will be a fixed-size buffer (16 elements, including the head and tail magic strings). But the values inside of it will be **strings** - and the buffers for the strings will all be allocated with `malloc()`.

Since the string buffers are allocated with `malloc()`, you will also need to use `free()` to release them - and we will also use `valgrind` to check your program for errors.

Finally, we'll require, in this project, for you to write your first Makefile.

1.1 Standard Requirements

- Your C code should adhere to the coding standards for this class:
  `https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en`
- 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. (Sometimes the error is serious - meaning "the operation cannot be performed." Sometimes, it's more minor, such as "two files are different" or "minor issues happened.")

  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 different 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).

2 New Requirement: `free()` all `malloc()`s

Starting with Project 5, you must now `free()` every buffer that you `malloc()`, before your program terminates. If you don't, then `valgrind` will report errors - and you will lose partial credit on your test-case.

This is, of course, a good habit to have! Always keep track of all memory that you have allocated with `malloc()` - and always be responsible to free it.

However, in the Real World, people often ignore the need to free memory, if the program is about to die. As I've mentioned in class, when your process dies, the OS will automatically free **all** of your memory - so it doesn't really matter whether you `free()`d everything or not.

But since we want you to practice `free()`ing your memory, we will require that you `free()` every buffer that you `malloc()`ed - even if you don't free it until right at the **end** of your program. (This is, probably, exactly what you'll do in Project 5!)

3 Turning In Your Code

Turn in your code using Gradescope. Turn in the following files:

```
rollingStrings.c
Makefile
```

Please submit them as two separate files; do not zip them up into a single zip file or tarball.

4 Grading

In this project, we have some new elements which we'll be grading.

- We will use `valgrind` to confirm that your program has no memory errors.
- You must provide a Makefile, which compiles your code.

We'll give details below.

4.1 Testcase Grading

I 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.[1]
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors (including failing to `free()` something before your program ended.

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

5 `Makefile` - Requirements

In this project, you must provide your own Makefile. The Makefile must include a rule which builds your executable (`tokenize`) from your source file (`tokenize.c`). In addition, your Makefile may (but is not required to) include other rules; common rules include `all` and `clean`.

Your rule must:

- Automatically build your executable from your `.c` file when we type

  ```
  make [programName]
  ```

  in the directory.
- Must automatically re-build the executable if the `.c` file is modified.
- **NOT** rebuild the executable if the `.c` file has not been modified.
- Use `gcc` as the compiler.
- Pass the `-Wall` option (because you always do!).
- Pass the `-g` option (so that we can run your executable under `valgrind`).

6 Grading

We have provided compiled version of the program for you; you should download them and run them (on Lectura) for comparison. The program can be copied from `/home/marahman/cs352/projects/proj05-2/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.). Note that you can copy the files using the `scp` command. You can find more how to use the `scp` command using the man page. One example to download the files to your current working directory of your non-lectura Linux machine:

```
scp lectura.cs.arizona.edu:/home/marahman/cs352/projects/proj05-2/* .
```

[1] Remember, we handle `stderr` more loosely than `stdout`. See below.

(We have provided the same on Piazza; however, realize that these will only run on a Linux machine running the x86-64 architecture. If don't have a linux machine, you should work on Lectura and only copy the files to your lectura account and your choice of working directory.)

We have also provided our grading script. (We'll do this for the first couple of assignments, in order to help you understand the requirements - but we will stop doing this as the programs get more complex!) You can copy it from `/home/marahman/cs352/projects/proj05-2/grade_proj05-2` on Lectura, or from the web.

Note that the grading and testing scripts are executable binaries but only have read permissions! Once you download or copy them to your working directory, you need to modify their permissions to execute them. You may need to review the UNIX lecture from Week 1 and are welcome to use google to see how you can change permissions.

Finally, we have provided a few example inputs to test your program with. (We will use additional ones when we grade your program.) **You should write additional test-cases** - try to be creative, and exercise your program in new ways.

NOTE 1: Since this project has multiple programs, it needs testcases for each one. The testcases are named `test_<progName>_*`. In order for the grading script to see your new testcases, please name them according to this standard.

NOTE 2: Some of the programs in this project use `stdin` as their input; name those testcases `test_*.stdin` . Some use command-line arguments; name those testcases `test_*.args` .

6.1 Exact Output

In each Project this semester, most of your grade will come from automatic testing of the code. You must match `stdout` **EXACTLY**, byte for byte. Common mistakes include:

- Extra or missing spaces
- Extra or missing blank lines
- Misspelled words
- Different capitalization

Your code must also give the same return value (0 or 1) as the example executable in every case.

`stderr` will be handled a little more loosely. In each testcase, we will check to see if the standard executable reported **some** error message to `stderr` or not. If it did, then your program must as well; if not, then your program must not print anything, either. However, we will not be comparing the exact error messages.

NOTE: Each testcase either passes, or fails entirely. We do not give partial credit for any testcase - although you may, of course, pass some testcases but not others.

6.2 Running the Grading Script

To run the grading script, make sure you place the following files in the same parent directory (e.g., `proj05-2`) and then run `./grade_proj05-2`

```
grade_proj05-2

example_*

test_*
[any input files required by the testcases]

Makefile
rollingStrings.c
```

The grading script does a number of things:

- Confirms that we have an example executable for each program.
- Confirms that each of the required files has the proper name, in a directory with the proper name. If not, you lose all points for that program.
- Checks that your Makefile works properly.
- Compiles your code. If your code doesn't compile at all, then you lose all points for that program. If it compiles but has warnings, then you will get a heavy deduction.
- Runs your code against all of the testcases. In each case, it runs both your code, and also the example executable. It checks to make sure that both have the same return code - and then also checks to make sure that the outputs match. If not, then it will give you a zero for that testcase. It also checks to make sure that your program runs cleanly under `valgrind`.

  (If you have no testcases for some program, then the script will give you a zero for that program.)

At the end, the grading script reports a score; this score is determined by the number of testcases that you passed - modified for any deductions.

6.3 Hand Grading

In addition, each program will be looked at by a human, to check for things which we can't automatically check, like:

- Some of the details of the spec
- Good comments
- Good indentation
- Reasonable variable names

6.4 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

6.4.1 Score Distribution

If any program compiles and runs, but warnings were produced by the compiler, you will lose half your score for that program.

- 10% - `Makefile` works properly
- 60% - `rollingStrings` testcases

The last 30% of the score will come from hand grading by the TAs.

7 Program 1 - Rolling Strings

This program reads a set of strings, using `malloc()` to allocate the buffer for each one. It keeps a set of them in a fixed-size array of pointers, and keeps the array sorted. As the array fills up, it will start discarding strings, and `free()`ing the memory involved.

Name your file `rollingString.c`

7.1 Input

This program reads its parameters from the command line. Each parameter is a file name; your program will open each file in turn, read the contents, and close it when you are done (see below).

The contents of each file will alternate between positive integers and words. Read the integers with `%d`, and the words with `%s`; this means, of course, that the words will be delimited by whitespace.

Each integer is the length of the largest string you might read in the next step; you must `malloc()` a `char` array of that size (plus space for the null terminator). Normally, each string will fit into the buffer that you've allocated - but in some testcases, it will be too long. You **must** limit the length that you read with `fscanf()` so that you do not overflow the buffer that you have `malloc()`ed.

7.2 Required Function: `readOneStr()`

You must declare a function matching the following prototype:

```
char *readOneStr(FILE *fp, int *err);
```

The job of `readOneStr()` is to read one integer (length), and then the string associated with it. It must handle `malloc()`ing the memory, and it must return the buffer that it allocated. (It must return `NULL` if it hits EOF.)

`readOneStr()` will also handle all input errors (see below). It will print out the "OOPS" messages if required - perhaps many of them. It will keep reading, as long as it takes, until it either successfully reads one string (which it returns) or it hits EOF (in which case it returns `NULL`.)

The `FILE*` parameter is (of course) the file to read from.

The `int*` parameter is an **"out"** parameter: you must write 1 to the `int` (not the pointer!) if an error occurs. (Do not modify the `int` if no error occurs.)

This works a little like `scanf()`, from the perspective of `main()`; `main()` should have a local `int` variable, and it should call `readOneStr()` like this:

```
int hasAnErrorOccurredYet = 0;
...
char *stringThatGotRead = readOneStr(currentFile, &hasAnErrorOccurred);
```

While `readOneStr()` does all of the `malloc()`s, it is the job of `main()` to keep track of the rolling buffer, and to `free()` buffers when required.

7.3 `malloc()` Buffer Size; `fscanf()` and `sprintf()`

You must `malloc()` a buffer which corresponds to the integer you've been given. Make sure to add space for the null terminator - but you **must not** intentionally allocate any "extra space" other than that.

This means, of course, that the format string that you provide to `fscanf()` cannot possibly be anything that you hard-coded; you **must** generate the string dynamically, based on the length you've been given. However, remember that the format string is no different than any other string - meaning that you can create an array of `char` and fill it up with text, and then pass it as the parameter to `fscanf()`.

To generate the format string, read the man page for the function `sprintf()`. `sprintf()` is a function just like `printf()` or `fprintf()` - except that instead of printing to `stdout` (or some other `FILE*`), it prints **into** a buffer that you've provided.

7.4 The Rolling Buffer

As you read strings from the input file(s), you will fill a fixed-size buffer. The buffer must be an array of strings, and msut be allocated as a local variable in `main()`. To allocate an array of strings, use the following syntax:

```
char *arrayOfStrings[SIZE];
```

The first and last elements must be certain constant strings (see the example output). Set these using **string literals**, not using `malloc()`. That is, your code to initialize the array should have something along the lines of:

```
myCoolArray[0] = "SOME_MAGIC_STRING";
```

Remember that if you try to `free()` memory which is a string constant, you will get an error from `valgrind` (and maybe even without `valgrind`).

Your array must have exactly 16 slots (including the first and last elements, which are special). Initialize all 14 of the "working" slots to `NULL`. As you read strings from the input files, insert them into the array, in order (compare the order of strings using the C standard library function `strcmp()`).

Until the array is full (that is, no more `NULL`s), you must not discard any of the strings. However, once the array is full, your code will then follow the same rule as `intRollingBuffer` did:

- First, discard the first element in the array (not counting the special string in slot 0)
- Then, insert the new element, keeping the array in order

After each new word that you read in, print out the entire contents of the array; see the example executable for the exact output format.

7.5 Files That Cannot Be Opened

If your program cannot open one (or many) of the files listed on the command line, print an error message to `stderr`, but continue running the program. Exit with a status of 1 if this happened at any point in your program's run.

7.6 Input Errors

If you are reading a file and you encounter any invalid input inside it, you will print out an "OOPS" message (see the example executable for the exact format). (Print these messages to `stdout`.) When your program ends, exit with a status of 1 if this happened at any time.

Check for the following input errors:

- You attempt to read one of the length fields (integers), but the input is non-numeric.

  Note: The output for this OOPS message prints the bad character twice: once as a two-character hex value, and once with `%c`. Don't worry about writing any special handlers for non-printable characters; just let `%c` print out whatever you read.

- You successfully read an integer, but the integer is not positive.
- You read an integer (indicating that a string should follow) but then hit EOF instead of finding a string.

Note that you should **NOT** consider a too-long-for-the-buffer string an error. (Often, the side effect of this will be a non-numeric error right afterwards; print out the non-numeric error if it happens.)

7.7 `malloc()` Failures

As always, check the return code on `malloc()`, and have some sort of error handling routine for it. However, as we've discussed, it won't be practical to test this code.

7.8 `fclose()`

When you are done with each file, you **must** call `fclose()` on it. If you don't, it's possible for your program to run out of "file handles." (The number of open files allowed to a program varies from OS to OS - but on Linux, I believe that the limit is 256, including `stdin, stdout,` and `stderr`.

To check that this is working properly, I'd encourage you to write a testcase which has the same file as input - hundreds of times.

See the man page for `fclose()` to see what its parameters are.

7.9 Other Outputs

The example executable prints out a few messages, as it runs, in order to show what's going on in the program. Check that program carefully, and make sure that your program prints out the same messages.

7.10 Exit Status

As noted above, there are two types of error conditions:

- Cannot open a file (print error message to `stderr`)
- Bad file contents (print an "OOPS" message to `stdout`)

Neither type should terminate your program early.

When your program terminates, exit with a status code of 1 if any type of error occurred; exit with a status code of 0 if not.