Project Strings + Performance

Due: October 24, 2017 at 11:59pm

1 Overview

This assignment was previously two separate projects — Strings and Performance — that for logistical reasons were combined into a single project with two parts.

You should approach each of the two parts as its own project, independent of the other part.

In the Strings portion of this project, you will implement a subset of C's string manipulation library.

In the Performance portion, you will optimize the performance of a function using techniques you learned in class.

2 Install the Stencil

To get started, run the following command in a terminal:

```
cs0330_install stringsperf
```

This will install the stencil code in your `course/cs0330/stringsperf` directory.

The only files you should edit are `strings/strings.c` and `perf/3matmult.c`.

3 Part 1: Strings

As cute as Boo is, she is still too young to speak complex sentences! For us normal folk, we would like to translate her blabs and exclamations to complete, sophisticated sentences. Luckily, we can get her spoken words to text, but we don't have a way of getting them to the Shakespearean English we want. If only we had a string library to accomplish this...

String manipulation is an important concept in computer science, and it is something that comes up very often in systems programming. Most programming languages have a string library that relieves programmers from writing their own string operations for every program. The C Standard Library has some excellent string manipulation facilities, but we know that Boo deserves a set of functions that she can really call her own!

Let's help translate Boo's blabs into some worldly remarks.

3.1 Assignment

For this part of the project, you will implement a subset of C's standard library string functions. You will use these functions in the upcoming Shell assignments to tokenize and search strings.

You should implement the following functions, all of which are documented extensively in the strings.c stencil file. As a hint, you may need to reuse earlier functions in later functions, so it might help to write these functions in order.

Your implementations should be efficient, but do not optimize your code at the expense of readability. (You will get a chance to write highly optimized code in Performance!) Please comment your code and explain any complicated logic. Your implementations should be at least as fast as the baseline times listed in Section 3.3 for full credit.

Here are the functions you should write, along with example uses of each:

- **`size_t strlen(const char *s);`**
  `strlen` computes the length of a string, excluding the terminating null byte.

```
size_t len = strlen("CS 33 Rocks!");  // len = 12
```

- **`size_t strspn(const char *s, const char *accept);`**
  `strspn` computes the number of bytes in the largest prefix of s that contains only characters from accept.

```
char *s = "we all love strings";
char *accept = "we alove";
size_t span = strspn(s, accept); // span = 12
```

- **size_t strcspn(const char *s, const char *reject);**
  strcspn computes the number of bytes in the largest prefix of s that contains only characters not in reject.

```
char *s = "coming up with example strings gets hard";
char *reject = "breakingthe4thwall";
size_t span = strcspn(s, reject); // span = 3
```

- **int strncmp(const char *s1, const char *s2, size t n);**
  strncmp is similar to the behavior of **strcmp**, which you do not have to implement for this assignment. **strcmp** compares two strings and returns a number less than, equal to, or greater than 0, if **s1** is found to be lexicographically less than, equal to, or greater than **s2**.
  strncmp is similar, except that it only compares the first (at most) **n** bytes of **s1** and **s2**.

```
int result = strncmp("same", "samexyz", 4); // result = 0
```

- **char *strncpy(char *dest, const char *src, size t n);**
  strncpy is similar to the behavior of **strcpy**, which you do not have to implement for this assignment. **strcpy** copies the contents of **src** into **dest**
  strncpy is similar, except that it only copies the first (at most) **n** bytes from **src** into **dest**.

```
char dest[] = "abcdefg";
strncpy(dest, "xyzt", 3); // dest now becomes "xyzdefg"
```

  Hint: Think about why we might have used a `char[]` instead of a `char*` for `dest`.

- **char *strstr(const char *haystack, const char *needle);**
  strstr finds the first occurrence of the string needle in the string haystack.

```
char *needle = "33";
char *haystack = "---32---33---33---";
char *location = strstr(haystack, needle);
// location = "33---33---"
```

- **char *strtok(char *str, const char *delim);**
  strtok returns a pointer to the first segment of str that does not contain any characters in delim. **strtok** should return non-empty token strings or **NULL** if there are no more

tokens in **str**. Note that **strtok** is stateful; to extract a sequence of tokens from the same string, you must make multiple calls by specifying the corpus string in the first call and **NULL** each time after.
Hint: **strtok** makes no promise to leave str untouched. You may overwrite characters with null terminators if you'd like.
Hint: The **static** keyword may be useful when writing **strtok**.

```
char *delim = "-";
// string is a char * that equals "---I---Love---33---"
char *token1 = strtok(string, delim); // token1 = "I"
char *token2 = strtok(NULL, delim); // token2 = "Love"
```

Remember that in C, you can think of the type **char *** as any of three things: a pointer to a character, a string, or an array of characters. These are all the same! Because of this, you can index into a string like an array.

3.2 Allowed Library Functions and Resources

Read the man pages for more information about each of the string functions if you're confused by the explanation, or come to hours and discuss the functions in more detail. All functions that you'll need to use are defined in **strings.c**. **There are no external functions allowed**.

3.3 Testing

We have included a few tests in the main function of the **tests.c** file. These tests will test the expected functionality of this string library. While they do cover basic functionality, we encourage you to write more of your own tests so that you can be sure that your functions work correctly before using them to implement subsequent functions.

Please do not modify the tests that are already in the file, as this will only make it harder for you to confirm that everything is working. Also, we will test your code with no compiler optimizations, so do not use a compiler flag to improve your times.

To build the test executable, run:

make

To test your work against the entire test suite, run:

./run_tests [number of repetitions] all

To test your work against a specific test or list of tests, run:

./run_tests [number of repetitions] <test name(s)>

We will be testing your code's efficiency with one million repetitions. If you do not specify a number when running the tests, it will default to one million.

Here are some reference times you should shoot for. Each time is in milliseconds.

	Baseline	Optimized	System
`strlen`	824ms	199ms	86ms
`strspn`	2s 339ms	124ms	93ms
`strcspn`	2s 365ms	141ms	99ms
`strncmp`	1s 956ms	199ms	177ms
`strncpy`	490ms	302ms	287ms
`strstr`	487ms	149ms	32ms
`strtok`	371ms	224ms	177ms

For full credit, we are looking for times that run faster than the baseline. If you want a challenge, see if you can reach the optimized times! The system column shows the times of the actual system functions.

4 Part 2: Performance

Roz, her undercover duties being fulfilled, wants to hone her already excellent math skills, in an effort to become the fastest math monster around. She has decided to speed things up, starting with calculating matrix multiplication faster and better.

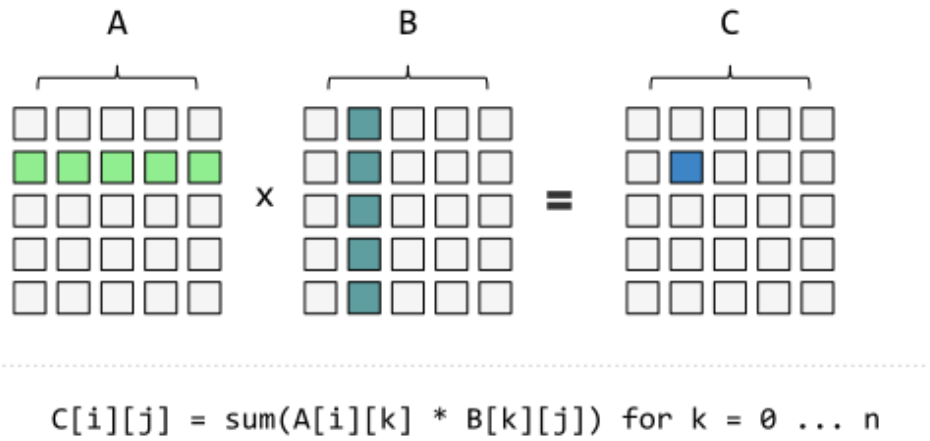
4.1 Assignment

The purpose of this assignment is to utilize various techniques to optimize the speed of C code. You will be writing and optimizing the performance of code that calculates the product of 3 matrices and explaining the techniques that you used.

Your task is to identify and implement ways to improve the function's performance. Please note before you begin that you must not change any of the code specified '**`DO NOT EDIT`**' in the stencil, as this is used for testing purposes.

4.2 3matmult

The code provided in **3matmult.c** sets up three matrices: **mul1**, **mul2**, and **mul3**. It is your job to calculate the product of all three matrices (i.e. **mul1 * mul2 * mul3**). Note that these matrices are all square, and of the same size (side length is **N**). The product of two matrices is calculated as follows: the value at position **(i, j)**, where **i** is the row, and **j** is the column, is equal to the dot product of row **i** in the first matrix and column **j** in the second matrix. The dot product is calculated by summing the product of each corresponding element, i.e. $r_0*c_0 + r_1*c_1 + ... + r_{N-1}*c_{N-1}$, where r_n and c_n represent the n^{th} element of the row and column being dotted, respectively. While this project does require the multiplication of 3 matrices, here is a refresher on matrix multiplication between two matrices (visually):



If you would like to declare a new matrix, use the following format:

```
double <matrix_name>[N][N] __attribute__((aligned(64)));
```

Aside: Our code makes use of 2 keywords, **__attribute__((aligned(64))**, to declare each matrix. You ***should not*** change this code, but here is an explanation of what these keywords do. The **__attribute__** keyword allows the user to specify special attributes to a variable, attributes used by the compiler when compiling. The **aligned** keyword forces the compiler to ensure that each matrix will be allocated and aligned with at least a 64 byte boundary. In this case, this allocation and alignment improves cache performance. For more information, see the gcc documentation.

4.3 Hints

Recall all of the techniques we've discussed in class. A combination of techniques is what will lead you to the best possible performance. Consider a few useful ones, reviewed below.

- Loop unrolling: this technique is invaluable in computations involving loops, but think carefully about how much unrolling makes sense.

- Short-circuiting: the basic implementation would first multiply **`mul1 * mul2`**, and then multiply that result by **`mul3`**. Is there a way to improve this?
- Cache optimization: variables are put into registers, which improves performance of the cache. However, the basic implementation would use few variables, opting instead to perform calculations by accessing many disparate locations of the matrices in question and multiplying those.

4.4 Testing and Grading for Performance

You only need to worry about filling in **`3matmult.c`**. After building your code, you can run

```
time ./3matmult
```

This will output the time it took for your code to execute on the matrix we've provided. We will be using the "**`user`**" time to test your code. For more thorough testing, we have provided our test script, which we will be using to test the performance of your code. You can run it with:

```
./time_test.sh <num_tests_to_run> <output_file>
```

As is stated earlier in the handout, it is imperative that you **do not modify** the existing code in the stencil, as this sets up the variables and tests your code for correctness. **`mul1`** is a matrix generated at the beginning of main, and **`mul2`** and **`mul3`** are the identity matrix, so the result of your multiplication should be a matrix equal to **`mul1`**. This is what is tested at the end of **`main`**. If all values match, the message **`matrices equal`** will be printed to standard out. Otherwise, the first disparity will be printed out, and the program will terminate. Also, as with Strings, we will test your code with our original Makefile, so do not add or change a compiler flag to improve your times.

It is important to note that failure to multiply all three matrices together (**`mul1 * mul2 * mul3`**) will result in *zero functionality and performance points*.

You will be graded on this project based on the following categories:

- **Functionality**
    - Your code should correctly compute the product matrix of **`mul1`**, **`mul2`**, and **`mul3`** and store it in **`res`**.
    - **Note**: As we specified above, the resulting matrix should be equal to **`mul1`**, but you **should not** simply return **`mul1`**, because your code should be accurate no matter what values are in **`mul1`** and **`mul2`**. Please do not simply assign **`res = mul1`**; you will not receive credit for this problem if you do not perform the computations.

- **Performance**
    - The faster your program runs, the more points you will receive.
    - You will not receive any performance points if your code does not produce correct output.
    - Most of the points for the Performance Part of this assignment will be given for the techniques you use to optimize your code. These points are *not* related to any actual runtime (in seconds) and will be given based on their conceptual merit.
    - 8% (8 points of 100 total assignment points) of **`stringsperf`** is dedicated to the runtime of your 3matmult implementation. Here are the 3matmult cutoff grade times:

Points	**`user`** Runtime of 3matmult
0/8	Greater than 20 seconds
2/8	19 seconds - 15 seconds
4/8	14 seconds - 10 seconds
6/8	9 seconds - 5 seconds
8/8	Less than 5 seconds

- **Explanation**:
    - You should explain all optimizations that you made in your *README*. This will be a significant portion of your grade, so make sure to explain what optimizations you made and why they make your program faster. If you considered any alternatives, explain why you chose to implement your function the way you did and what tradeoffs you may have made.

5 Handing In

To hand in your project, run the command:

```
cs0330_handin stringsperf
```

from your **`stringsperf`** directory.

Your handin must be contain two directories — **`strings`** and **`perf`** — that separate your work for the two parts of this project. The directories should be organized as follows:

- **`strings`** directory
    - All **`.c`** and **`.h`** files containing code you have written for the Strings portion of this assignment.
    - A *README* explaining your approaches to **`strstr`** and **`strtok`**. You should also document any known bugs and collaborators.
- **`perf`** directory
    - All **`.c`** and **`.h`** files containing code you have written for the Performance portion of this assignment.
    - A *README* explaining your optimizations, as described in the Performance section of the handout. You should also document any known bugs and collaborators.

If you wish to change your handin, you can do so by re-running the handin script. Only the most recent handin will be graded.

Important note: *If you have already handed in your assignment, but plan to hand in again after the TAs start grading at noon on Saturday, October 28th, in addition to running the regular handin script (***`cs0330_handin stringsperf`***), you must run* **`cs0330_grade_me_late stringsperf`** *to inform us not to start grading you yet. You must run the script by noon on 10/28*. **If you run this script, you will get grades back later than other students.**

If you do not run this script, the TAs will proceed to grade whatever you have already handed in, and you will receive a grade report with the rest of the class that is based on the code you handed in before we started grading.

If something changes, you can run the script with the **`--undo`** flag (before noon on 10/28) to tell us to grade you on-time and with the **`--info`** flag to check if you're currently on the list for late grading.

These instructions apply to all projects unless otherwise stated on the handout.