Thread-Safe, Reference-Counted String Heap

Extra credit project for CIS 415

Due at 11:59pm on Thursday, 8 June 2023

By now, you should be quite familiar with use of the C heap via invocations of `malloc()/free()` in `<stdlib.h>` to dynamically allocate storage while solving a problem. It is quite common for many of the heap allocations to be C strings, and is most easily achieved by calling `strdup()` in `<string.h>`.

Certain classes of applications, such as those that use Tables of records of string fields, have many records with identical strings in particular fields. Blindly using `strdup()` for each field will cause multiple copies of the same string to be allocated on the C heap. For very large tables, such a naïve approach may cause the program to exhaust the C heap.

A better approach would be to store a string once on the C heap, and for subsequent *allocations* of that string to simply return the pointer to that single copy; in order to use such an approach, we have to keep track of how many times the string has been *allocated* (reference count). Each call to free the string would decrement the reference count. Only when the reference count goes to 0 will the single allocation of the string on the C heap be freed.

For this project, you are to write **thread-safe** C code to implement `str_malloc()` and `str_free()` that deliver this reference-counted functionality.

1 The signatures for the string heap functions

The following header file, **strheap.h**, defines the signatures enabling a program to use the string heap.

```
#ifndef _STRHEAP_H_
#define _STRHEAP_H_

/* BSD header removed to conserve space */

/*
 * functions for a ref-counted string heap
 *
 * for use in programs that store multiple copies of the same string on the
 * heap
 *
 * a unique string will only be stored once on the heap, and a reference
 * count is kept of the number of times that string has been str_malloc()ed.
 *
 * each str_free() causes the associated reference count to be decremented.
 *
 * when the reference count reaches 0, the string is purged from the heap
 */

#include <stdbool.h>

/*
 * "duplicate" `string' on the string heap
 *
 * returns the address of the heap allocated string or NULL if malloc() errors
 *
```

```
 * increments the reference count for the string
 */
char *str_malloc(char *string);

/*
 * "free" `string' on the string heap
 *
 * returns true if free'd, or false if the string was not present on the string
 * heap
 *
 * if the decremented reference count has reached 0, the string is purged from
 * the heap
 */
bool str_free(char *string);

#endif /* _STRHEAP_H_ */
```

2 Constraints on your implementation

- It is essential that your implementation be thread safe; while you may not have realized it, the C heap **IS** thread-safe. Thus, any data structures used to implement the string heap must be sufficiently protected in a multi-threaded environment to prevent race conditions.
- Your code must initialize itself, either through global variable initialization, or runtime initialization on the first call to one of the string heap routines.
- If your program terminates by calling `exit()`, or by a return from `main()`, your system must return any heap allocated storage used to implement the string heap. In other words, if you run your program under `valgrind`, and it terminates normally, there should be no memory leaks due to your string heap implementation. You might want to consult the man page for `atexit(3)` to see how you might achieve this.

3 A simple implementation of strheap.c that is NOT reference counted

Since the C heap is thread-safe, we can trivially implement **non-reference-counted** versions of `str_malloc()` and `str_free()`. Here is the code:

```
#include "strheap.h"
#include <stdlib.h>
#include <string.h>

/*
 * functions for a string heap
 */

/*
 * "duplicate" `string' on the string heap
 * returns the address of the heap allocated string or NULL if malloc() errors
 */
char *str_malloc(char *string) {
    return strdup(string);
}

/*
 * "free" `string' on the string heap
 * returns true if free'd, or false if the string was not present on the string
 * heap
```

```
 */
bool str_free(char *string) {
    free(string);
    return true;
}
```

Obviously, this implementation does not address the multiple copies of the same string on the C heap issue.

4 Implementation

You must use bucket hashing for your string heap. The number of buckets must be doubled when the load factor in the table exceeds 5.0. With a good hash function, this should provide O(k) asymptotic complexity for string heap operations.

5 Test programs provided

I have provided three test programs that can be linked to your string heap implementation:

- test.c - a single-threaded program that accepts strings to store on the string heap from the argument list; its usage string is "./test [-d] string …"; if -d is specified, the program will free all of the occurrences of each string from the string heap; if not, it will simple str_malloc each string on the string heap and then exit normally.
- mthtest.c - a multi-threaded program that accepts strings to store on the string heap from the argument list; its usage string is "./mthtest string …".
- ttest.c - a program that reads in a CSV spread sheet, converting it to a Table ADT instance, manipulates the ADT instance, and then exits normally after destroying the table.

6 The starting archive

P4start.tgz contains the following files:

- strheap.h - the header file for the string heap functions
- strheapbase.c - the non-reference counted version shown in section 3 above
- test.c - the simple test program
- mthtest.c - the multi-threaded test program
- ttest.c - the table testing program
- table.h - header file for the Table ADT
- table.c - source file for the Table ADT
- row.h - header file for the Row ADT
- row.c - source file for the Row ADT
- *.csv - A number of csv files that can be used by ttest
- Makefile - a makefile that creates test, mthtest, and ttest from your strheap.c

Note that the makefile has a macro defined as "WHICHSOURCE=strheapbase". When you wish to make your test programs using your implementation in strheap.c, you will need to edit the makefile, replacing that macro definition by "WHICHSOURCE=strheap".

7 Submission

You will submit your solution electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named <duckid>-project4.tgz, where <duckid> is your "duckid". It should contain strheap.c and a document named report.txt, describing the state of your solution, and documenting anything of which we should be aware when marking your submission. No other files should be in your TGZ archive.

Within the archive, these files should **not** be contained in a folder. Thus, if I upload "jsventek-project4.tgz", then I should see something like the following when I execute the following command:

```
$ tar –ztvf jsventek-project4.tgz
-rw-r--r-- jsventek/group     8127 2016-10-30 16:37 strheap.c
-rw-r--r-- jsventek/group      325 2016-10-30 16:30 report.txt
```

Your source file must start with an "authorship statement", contained in C comments, as follows:

- state your name, your duckid, and the title of the assignment (CIS 415 Project 4)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

We will be compiling your code and thoroughly testing it. We will also be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

Grading Rubric

Your submission will be marked on a 50 point scale. Substantial emphasis is placed upon **WORKING** submissions, and you will note that a large fraction of the points are reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission.

You must be sure that your code works correctly on the virtual machine under VirtualBox, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission.

The marking scheme is as follows:

Points	Description
5	Your report – honestly describes the state of your submission
3	code compiles
2	code compiles with no warnings
2	‘./test -d one two three four five six seven eight nine ten’ runs correctly with NO memory leaks
3	‘./test one two three four five six seven eight nine ten’ runs correctly with NO memory leaks
5	‘./mthtest one two three four five six seven eight nine ten’ runs correctly with NO memory leaks
5	‘./ttest <t.csv’ runs correctly with NO memory leaks
5	‘./ttest <big.csv’ runs correctly with NO memory leaks
5	‘./ttest <huge.csv’ runs correctly with NO memory leaks
5	‘./ttest <enormous.csv’ runs correctly with NO memory leaks
10	code could have worked with minor modification to the source.

Note that:

- Your report needs to be honest. Stating that everything works and then finding that it doesn’t is offensive. The 5 points associated with the report are probably the easiest 5 points you will ever earn as long as you are honest.
- The points for “could have worked” is the maximum awarded in this category; your mark in this category may be lower depending upon how close you were to a working implementation.