clab

Willard Rafnsson

IT University of Copenhagen

This lab drills you in the C style of programming, which you need to be good at to solve the remaining labs in this course. Some of the skills tested are:

- explicit memory management, as required in C,
- creating and manipulating pointer-based data structures,
- working with strings,
- enhancing performance of key operations by storing redundant information in data structures,
- implementing robust code that operates correctly with invalid arguments, including `NULL` pointers.

Introduction

This lab involves implementing a queue, supporting both last-in, first-out (LIFO) and first-in-first-out (FIFO) queueing disciplines. The underlying data structure is a singly-linked list, enhanced to make some of the operations more efficient.

The file `queue.h` contains declarations of the following structures:

```
/* Linked list element */
typedef struct list_ele {
    char *value;
    struct list_ele *next;
} list_ele_t;

/* Queue structure */
typedef struct {
    list_ele_t *head; /* First element in the queue */
} queue_t;
```

These are combined to implement a queue of strings, as illustrated in Figure **??**. The top-level representation of a queue is a structure of type `queue_t`. In the starter code, this structure contains only a single field head, but you will want to add other fields. The queue contents are represented as a singly-linked list, with each element represented by a structure of type `list_ele_t`, having fields value and next, storing a pointer to a string and a pointer to the next list element, respectively. The final list element has its next pointer set to `NULL`. You may add other fields to the structure `list_ele`, although you need not do so.

Recall that a string is represented in C as an array of values of type `char`. In most machines, data type `char` is represented as a single byte. To store a string of length `l`, the array has `l + 1` elements, with the first l storing the codes (typically ASCII[1] format) for the characters and the final one being set to `0`. The value field of the list element is a pointer to the array of characters. The figure indicates the representation of the list `["cab","bead", "cab"]`, with characters a–e represented in hexadecimal as ASCII codes 61–65. Observe how the two instances of the string `"cab"` are represented by separate arrays—each list element should have a separate copy of its string.

In our C code, a queue is a pointer of type `queue_t *`. We distinguish two special cases: a `NULL` queue is one for which the pointer has value `NULL`. An empty queue is one pointing to a valid structure, but the head field has value `NULL`. Your code will need to deal properly with both of these cases, as well as queues containing one or more elements.

[1] "American Standard Code for Information Interchange," developed for communicating via teletype machines.

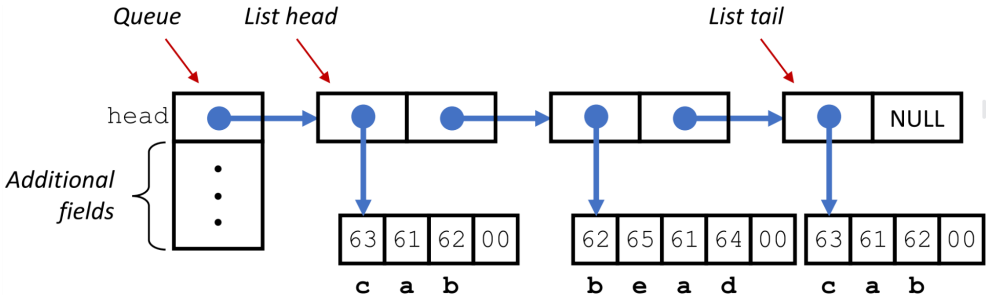

Figure 1: Linked-list implementation of a queue. Each list element has a `value` field, pointing to an array of characters (C's representation of strings), and a `next` field pointing to the next list element. Characters are encoded according to the ASCII encoding (shown in hexadecimal.)

Task

Your task is to modify the code in `queue.h` and `queue.c` to fully implement the following functions: `queue_new`, `queue_free`, `queue_insert_head`, `queue_insert_tail`, `queue_remove_head`, `queue_size`, and `queue_reverse`. More details can be found in the comments in these two files, including how to handle invalid operations (e.g., removing from an empty or `NULL` queue), and what side effects and return values the functions should have.

For functions that provide strings as arguments, you must create and store a copy of the string by calling `malloc` to allocate space (remember to include space for the terminating character) and then copying from the source to the newly allocated space. When it comes time to free a list element, you must also free the space used by the string. You cannot assume any fixed upper bound on the length of a string—you must allocate space for each string based on its length. Note: `malloc`, `calloc`, and `realloc` are the only supported functions in this lab for memory allocation; any other functions that allocate memory on the heap may cause you to lose points.

Two of the functions, `queue_insert_tail` and `queue_size`, will require some effort on your part to meet the required performance standards. Naive implementations would require $O(n)$ steps for a queue with n elements. We require that your implementations operate in time $O(1)$, i.e., that the operation will require only a fixed number of steps, regardless of the queue size. You can do this by including other fields in the `queue_t` data structure and managing these values properly as list elements are inserted, removed and reversed. Please work on finding a solution better than the $O(n^2)$ solution for all of the functions.

Your program will be tested on queues with over 1,000,000 elements. You will find that you cannot operate on such long lists using recursive functions, since that would require too much stack space. Instead, you need to use a loop to traverse the elements in a list.

Prepare

Study C to the point where you feel comfortable with the skills being tested. There are many good resources on this; we recommend Kernighan and Ritchie's *The C Programming Language* (second edition). Review how linked lists work, and what big-O notation is (i.e. what $O(n)$ means). Finally, the authoritative source on library functions are manual pages; run `man FUN` for e.g. `malloc`, `free`, `sizeof`, `strlen`, `strcpy`, and `strncpy`.

Build, Run, Test

The handout uses GNU make for build automation.

Build (i.e. compile):

```
$ make
```

This creates `qtest`, a command-line interpreter for experimenting with your queue implementation.

Run `qtest`, or:

```
$ make run
```

Type `help` to see a list of available commands.

Test your implementation by running `driver.py`, or:

```
$ make test
```

Clean all generated files:

```
$ make clean
```

Files

Queue API. You will modify and hand in (only) these two files.

- **`queue.h`:** declarations for the queue API.
- **`queue.c`:** implementation of the queue API.

Queue API tester:

- **`qtest.c`:** command-line interface & interpreter for testing the Queue API. It uses the following:
- **`console.{c,h}`:** implementation of a command-line interpreter
- **`report.{c,h}`:** implementation of printing of information at different levels of verbosity
- **`harness.{c,h}`:** customized version of `malloc`/`free`/`strdup` to provide rigorous testing framework

Test files & test driver:

- **`driver.py`:** the driver; runs `qtest` on a standard set of traces
- **`traces/trace-XX-CAT.cmd`:** the trace files used by the driver. These are input files for `qtest`.
  - Study them; see what tests are being performed (they are short & simple).
  - XX is the trace number (1-15). CAT describes the general nature of the test.
- **`traces/trace-eg.cmd`:** a simple, documented trace file to demonstrate the operation of `qtest`

Project files:

- **`README`:** how to get started.
- **`README.md`:** important URLs, and handin instructions.
- **`Makefile`:** builds & runs `qtest`, and runs `driver.py`

Driver

The driver program `driver.py` runs `qtest` on the trace files to test your code. To aid your debugging, `qtest` is compiled with AddressSanitizer. This is an instrumentation tool similar to Valgrind that detects common memory issues such as accessing invalid memory, calling `free()` multiple times on the same address, or leaking memory. Programs are instrumented with AddressSanitizer at compile time, so you do not need to do anything extra to take advantage of it. However, be aware that Valgrind and AddressSanitizer cannot be used at the same time.

When AddressSanitizer detects an error, it will print an error similar to the following, then exit the program:

```
==4662==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020
00000318 at pc 0x0000004e37e2 bp 0x7ffdbf037f30 sp 0x7ffdbf0376e0
READ of size 1025 at 0x602000000318 thread T0
#0 0x4e37e1 in __asan_memcpy (<...>/qtest+0x4e37e1)
#1 0x52e251 in queue_remove_head <...>/queue.c:160:9
#2 0x5276f9 in do_remove_head <...>/qtest.c:273:16
#3 0x52c0ae in interpret_cmda <...>/console.c:224:14
#4 0x52be0a in interpret_cmd <...>/console.c:251:15
#5 0x52ca4a in cmd_select <...>/console.c:635:9
#6 0x52da21 in run_console <...>/console.c:722:9
#7 0x52876a in main <...>/qtest.c:527:16
#8 0x7fc4eb07eb96 in __libc_start_main /build/glibc-OTsEL5/glibc-2.
27/csu/../csu/libc-start.c:310
#9 0x41a829 in _start (<...>/qtest+0x41a829)
```

This list of function names is called a *stack trace*, representing the execution state when the error occurred. The lowest numbered frame `#0` indicates the innermost function call: for instance, we can see that the error was caused by `__asan_memcpy`. However, the first function that is actually in our code is the frame `#1`. Therefore, we should look at the referenced location `queue.c:160:9` to find the error, which is referring to line 160 of `queue.c`.

Evaluation

Autograder

Your program will be evaluated using the fifteen traces described above. You will be given credit (either 6 or 7 points, depending on the trace) for each one that executes correctly, summing to a maximum score of 100. This will be your score for the assignment—the grading is completely automated.

Style

This lab will not be style graded. However, the TAs will evaluate the coding style of your submission, and may choose to reject your handin if it does not follow a known style consistently. Consult the course material for tips on coding styles, pick one style, and follow it.

Memory Safety

Memory safety issues are a serious correctness issue, so you should make sure to fix any errors that are detected by AddressSanitizer. If you need help interpreting the output, feel free to ask for help on Slack.

Handin & Passing

Instructions for when, where, and how, to hand in, as well as passing criteria, are available on the course website.

Logistics

This should be a straightforward lab for students who are fully prepared to take this course. A well-prepared student can complete this lab in 2-3 hours; you will need longer if you struggle with computer literacy or C.

Start early enough to get this lab done before the due date. Assume things will not go according to plan; allow extra time for dropped Internet connections, corrupted files, traffic delays, minor health problems, etc.

We expect you to work on the assignment on `cos`. If you work on the assignment on your own computer instead, and then encounter technical problems, then we may be unable to help you; we will do our best to help troubleshoot your own setup, but can only guarantee support on `cos`. To check if a technical problem is caused by your own setup, first try doing what you were trying to accomplish on `cos`, to see if you encounter the same problem there.

If you found you had trouble writing this code or getting it to work properly, this may be an indication that you need to upgrade your computer literacy or C programming skills over the next month. Starting with Lab 3, you will need to write programs that require a mastery of the skills tested in this assignment.

A good place to start is to carefully study Kernighan and Ritchie. This book documents the features of the language and also includes a number of examples illustrating good programming style. The book is a bit dated, and thus does not contain some more modern features of the language, such as the `bool` data type, but it is still considered one of the best books on how to program in C.

This is an individual project. No aids (be it fellow classmates, other people, blogs & other websites, or AI-powered language models) for producing a solution are permitted (so, queries like "linked list implementation in C" are off limits). For further details on this, please review ITU's policy on academic integrity.

Acknowledgment

This lab is a (barely) modified version of the C Programming Lab from the *Introduction to Computer Systems* course at Carnegie Mellon University, with some inspiration from changes made by Jim Huang for *Linux Kernel Design/Implementation* at National Cheng Kung University. All praise to them (any flaws are surely mine).