Introduction

The second assignment will focus on different ways to implement FIFO. You will also develop a proper GitHub project for this assignment, complete with test harness, documentation, README, and Makefile. The assignment is due on September 18 at 9:00 am Mountain time. You will be provided with a GitHub classroom link to accept and create your git repo. Your submission repo will have multiple .c and .h files, test cases, and a Makefile. Details are below. Finally, you will create "**ready-for-grading**" tag BEFORE THE DUE DATE/TIME so as not to incur the late penalty.

You may consult with other students, the SAs, and the instructor in reviewing concepts and crafting solutions to problems (and may wish to credit them in documentation).

Background

A FIFO object—often called a *queue* instead—is intended to connect some code which *produces* data with some code which *consumes* data. This design pattern is relatively widespread; it is commonly used when producer and consumer threads run asynchronously with respect to each other.

One common use of a FIFO might be for bytes being read from a UART. In this example, the producer is likely to be the interrupt service routine (ISR) for the UART hardware. When a byte arrives at the UART and the interrupt is triggered, the ISR will enqueue the byte onto a FIFO. Meanwhile, a consumer thread will periodically wake up and dequeue bytes that have arrived. Most likely, the consumer thread will run far less frequently than the producer thread, but it will process more bytes of data at a time.

For this assignment, you will implement two objects:

a) `llfifo`: A linked-list implementation of a FIFO, suitable for video or other applications.

b) `cbfifo`: A circular-buffer implementation of a FIFO, suitable for UART.

Details for each of the implementations follow.

Assignment Detail

Throughout this assignment I am using the following terms:

- "Capacity" refers to the total size of the FIFO – how many elements could it hold, given its current memory usage?
- "Length" refers to the number of elements currently enqueued on the FIFO

With these definitions, it follows that 0 <= length <= capacity.

`llfifo`: Linked-list FIFO

The `llfifo` implementation will be capable of growing dynamically through an efficient use of malloc(). The caller will suggest an initial capacity when the FIFO is created, but if the caller subsequently attempts to enqueue more elements onto the FIFO than the current capacity would support, `llfifo` will silently increase its capacity by one up to the limit imposed by max_capacity. For simplicity, the capacity of an `llfifo` instance will not shrink until the FIFO is destroyed.

The FIFO should implement the following API:

```
llfifo_t *llfifo_create(int capacity, int max_capacity);

int llfifo_enqueue(llfifo_t *FIFO, void *element);

void *llfifo_dequeue(llfifo_t *FIFO);

void *llfifo_peek(llfiof_t *FIFO);

int llfifo_length(llfifo_t *FIFO);

int llfifo_capacity(llfifo_t *FIFO);

int llfifo_max_capacity(llfifo_t *FIFO);

void llfifo_destroy(llfifo_t *FIFO);
```

Details for each of these functions may be found in the file llfifo.h.

Note that it is possible for a `llfifo` buffer node to be created, but to be currently unused. The API is designed this way because we want to use malloc efficiently—we don't want to call malloc() on every enqueue operation and call free() on every dequeue operation. Instead, your FIFO will need to have some means for keeping track of these already-created-but-currently-unused elements.

As an example, let's say the caller creates a new `llfifo` by calling `llfifo_create(5, 100)`. Immediately after this function returns, the `llfifo` will have created 5 buffer nodes internally. However, at this point, the FIFO is empty, so those 5 nodes are in this already-created-but-currently-unused state. The following code snippet extends the example:

```
llfifo_t *my_FIFO = llfifo_create(5, 100);
assert (llfifo_length(my_FIFO) == 0);
assert (llfifo_capacity(my_FIFO) == 5);
assert (llfifo_max_capacity(my_FIFO) == 100);
llfifo_enqueue(my_FIFO, "Sleepy");   // does not result in a malloc (len = 1, cap = 5)
llfifo_enqueue(my_FIFO, "Grumpy");   // does not result in a malloc (len = 2, cap = 5)
llfifo_enqueue(my_FIFO, "Sneezy");   // does not result in a malloc (len = 3, cap = 5)
```

```
llfifo_enqueue(my_FIFO, "Happy");    // does not result in a malloc (len = 4, cap = 5)
llfifo_enqueue(my_FIFO, "Bashful");  // does not result in a malloc (len = 5, cap = 5)
llfifo_dequeue(my_FIFO);             // removes "Sleepy"; len = 4, cap = 5
llfifo_enqueue(my_FIFO, "Dopey");    // does not result in a malloc (len = 5, cap = 5)
llfifo_enqueue(my_FIFO, "Doc");      // DOES result in a malloc (len = 6, cap = 6)
```

For initial capacity parameter of 0, create function should not error but still create the containing structure without the nodes. For any negative capacity, the return will be an error code. Similarly, max_capacity value must be greater than 0.

When `llfifo_enqueue` is called and the FIFO is already full (i.e., length == capacity), the capacity should be increased by one upto max_capacity. If the FIFO is already used up at max_capacity, it should drop the oldest element and replace it with the most recent enqueued data.

When error is encountered in allocating more nodes, you must free up all of the memory that was malloced prior to the error. Similarly, in `llfifo_destroy` function, you must free up all memory that was allocated.

It must be possible for the user to create and use multiple `llfifo` instances simultaneously. This means your implementation probably should not use global or static variables.

`cbfifo`: Circular Buffer FIFO

The `cbfifo` implementation will allow cbfifo creation, with a size of **256** bytes which is statically allocated. (You should make this constant a #define in your code.) The FIFO should implement the following API:

```
size_t cbfifo_enqueue(void *buf, size_t nbyte);

size_t cbfifo_dequeue(void *buf, size_t nbyte);

size_t cbfifo_peek(uint8_t *byte);

size_t cbfifo_length();

size_t cbfifo_capacity();
```

Details for each of these functions may be found in the file cbfifo.h.

It is important to realize that from the perspective of the `cbfifo`, the data being passed across the queue is simply a stream of bytes. The `cbfifo` code has no idea how to interpret those bytes; instead, interpretation of the bytes is up to the caller.

To explain this in an example, suppose a caller were to enqueue the 4 bytes of an integer, for instance:

```
int x = 310;
if (cbfifo_enqueue(&x, sizeof(x)) != sizeof(x)) {
    // handle error case of the cbfifo being full
}
```

Because the `cbfifo` code was handed a pointer and a size (which will be either 4 or 8 depending on architecture), the `cbfifo` would simply enqueue 4 or 8 bytes.

Now, later, a caller calls **`cbfifo_dequeue`**. Perhaps the caller does this:

```
int y;
assert(cbfifo_dequeue(&y, sizeof(int)) == sizeof(int));
// use y for something
```

But perhaps instead the caller is streaming this data over a UART, and needs to receive it byte by byte:

```
uint8_t byte;
while (cbfifo_dequeue(&byte, 1) == 1) {
   // send byte over UART
}
```

Both uses of `cbfifo` should work with your implementation. Also, ensure that you are not using any builtin, modulo, multiplication or division operators. You can use add/subtract/shift operations.

General Assignment Notes

Normally, considerable care must be taken to ensure that a FIFO implementation is thread-safe, since in the intended use, the producer and consumer threads could be calling the FIFO API simultaneously—or, conversely, perhaps one thread is an ISR that could run at any point. However, for this assignment we will ignore thread safety issues.

Again for this assignment, you may NOT use standard C library functions, including the printf family, in your production code. However, you may use such functions in any of the test code. (You *may* use malloc, free, memset, and memcpy in your production code.)

For each of the FIFO implementations, you may wish to create an additional function for debugging, called for instance “cbfifo_dump_state()”. This function may use printf() to output all the internal state of the FIFO. This function is not a requirement of the assignment, but might save you time in debugging.

Each FIFO should have a corresponding test function, for instance test_llfifo(). This function should thoroughly exercise the corresponding FIFO. It can either return 1 if all tests succeeded, and 0 otherwise;

or it can call assert() on a test failure (in which case, the function returning indicates that all tests succeeded). As before, try hard to ensure that the tests cover all the possible cases for the FIFO.

Your main() routine should call both test functions.

All functions should check for invalid inputs and return appropriate errors.

Your submission should include at least the following files:

```
llfifo.h*, llfifo.c
cbfifo.h*, cbfifo.c
test_llfifo.h, test_llfifo.c
test_cbfifo.h, test_cbfifo.c
main.c
Makefile* – A starter Makefile is provided. You should make any edits
such that the SAs can build your executable just by using your
Makefile.
README.md
```

(Files marked with a * are provided for you and should not be edited.)

Development Environment

For this project, you may again use the CUPES Linux server. As in Assignment 1, for this assignment **autograder2** is provided.

For automated tests, you can type following command on CUPES for using **autograder2**:

```
autograder2 main.c llfifo.c llfifo.h cbfifo.c cbfifo.h
```

Please include all your main .c and .h files for autograder 2. Your main function will be removed during autograding. Same restrictions apply for autograder as for the assignment 1.

Your code should follow the ESE C Style Guide (posted on Canvas) as closely as possible.

When compiling with GCC, use the -Wall and -Werror compiler flags at a minimum. Your code should have no compilation errors or warnings.

Grading

Points will be awarded as follows:

Points	Item
2	Did you submit the Git URL on Canvas(1 pt) and create 'ready-for-grading' tag(1 pt) in GitHub repo?
3	Compilation: Does the code compile cleanly with no errors, when run with -Wall -Werror? (Points are all-or-nothing)
3	Function documentation: Is each function well documented, including parameters, return value, and error codes?
2	Are #defines used appropriately?
5	Elegance/Readability: Is there good and consistent indentation? Are variable names appropriate and understandable? Does it otherwise follow the ESE Guidelines (excluding items already graded above, like comments)?
20	llfifo correctness: 5 points each for automated test scenarios with initial capacities of 1, 5, 20, and 0
5	llfifo multiple simultaneous instances: Based on automated test results
5	llfifo max capacity test: Based on automated test results
5	llfifo dynamic memory allocation: Are malloc/free used correctly? Based on automated test results
10	llfifo test coverage: Subjectively, are there good automated tests?
20	cbfifo correctness: Based on automated tests
5	cbfifo operations: No use of modulo, multiplication, division(All or nothing).
5	cbfifo static memory allocation: Does the cbfifo implementation statically initialize the circular buffer?(All or nothing)
10	cbfifo test coverage: Subjectively, are there good automated tests?

Remember, if you do a good job on your test cases, you will almost certainly get the code entirely correct, as well. Good luck!