Lab 8 - Description

(Implementing of a collection of queues in C)

Lab Overview:

For this lab, we will be learning how to use a circular ring buffer to implement queue and then extending it to implement a collection of queues. A queue is a linear data structure (like a stack) that follows the First-In-First-Out (FIFO) principal. In general, a queue consists of the following: A) A buffer, B) A pointer to the head of the queue, B) A pointer to the tail of the queue. As data is pushed into the queue, the tail pointer is pushed forward so that it is always pointing to the position next to the last element in the queue. This process of pushing data into the queue is known as enqueuing. On the other hand, when data is accessed and popped from the queue it occurs at the head of the queue. The head pointer is then pushed forward to keep up with the last element in the queue. This is known as dequeuing. Because of this behavior, Circular Ring Buffers (CRBs) are the main primitive data structure used in implementing a queue.

For this lab, we will first implement a single queue for the meal ticket struct. This will allow us to push/pop meal tickets onto/from the queue. Once we have completed this, we will implement a registry for meal tickets for a restaurant that contains a queues for the following course types: Breakfast, Lunch, Dinner, Bar. This will allow waiters to electronically push orders into the registry for the chefs to pull and complete.

Core Tasks:

1. Implement a single Meal Ticket Queue (MTQ) struct.
2. Implement the meal ticket registry.
3. Implement the enqueue() and dequeue() functions.
4. Add a main() that pushes meal tickets to each queue and displays the output.

Task Details:

Implement a single Meal Ticket Queue (MTQ) struct.

First, we will need a struct to hold the data for individual meal tickets. The meal ticket struct is as follows:

```
struct mealTicket {
        Int ticketNum;
        char *dish;
}
```

Fig 1: Meal ticket struct

Once we have this struct we can get started on the MTQ. First we will need a struct for the queue itself. This struct is as follows:

```
struct MTQ {
        char name[MAXNAME];
        mealTicket * const buffer;
        int head;
        int tail;
        const int length;
}
```

Fig 2: MTQ struct

Implement the meal ticket registry.

Since we now have the MTQ struct, next we need the registry. The registry is a simple collection of MTQ's. This can be done by the following: `MTQ *registry[MAXQUEUES]`

Implement the enqueue() and dequeue() functions.

From here you need to do the following:

1. Write the following method: `enqueue(char *MTQ_ID, mealTicket *MT)`
    - This method will push the meal ticket **MT** to the MTQ whose name is given by **MTQ_ID.**
    - The ticketNum is assigned by the enqueue function.
    - This method will increment the tail pointer each time it pushes data so that the tail pointer always points to the position after the newest element of the queue.
    - **Note:** MT is a filled mealticket to be pushed to the queue.
    - **This method must return 1 when successful else 0 when not successful.**
2. Write the following method: `dequeue(char *MTQ_ID, int ticketNum, mealTicket *MT)`
    - This method will pop and retrieve the mealTicket specified by **ticketNum** from the queue specified by **MTQ_ID**.
    - **Note:** For this function, **MT** is an empty meal ticket struct that we will copy data to.

- **This method must return 1 when successful else 0 when not successful.**

Add a main() that pushes meal tickets to each queue and displays the output.
Your main **must** do the following:

1. Initialize the registry (this is a global variable).
    a. Create MTQs with the following names: **Breakfast, Lunch, Dinner, Bar** and push them into the registry.
2. Create and push 3 meal tickets into each queue.
3. In a round robin fashion pop a meal ticket from each queue and display it until all queues are empty.
    a. You main **must** display the output as follows (**exactly**) :
    "Queue: <name> - Ticket Number: <n> - Dish: <dish> "
4. Your main must perform the following tests:
    a. Dequeue when a queue is empty
    b. Dequeue when a queue is full
    c. Enqueue when a queue is full
    d. Enqueue when a queue is empty

    Your main **must** print the output from each test case as follows:
    "Test Case: <A/B/C/D> - Result: <Success/Fail>"

Remarks:

This lab is pretty straight forward if you give it some thought. While this lab is directly related to the project there are some significant differences (i.e. the dequeue method is completely different and there is no get method).

Submission Requirements:

In order to receive any credit for a lab, completion of the labs' core tasks must be demonstrated to the TA's. In order to receive points for this lab the student must do the following:

1. Compile and run your code (we must see it compile and execute successfully).
2. Valgrind output with leak-check and mem-check showing no memory leaks.
3. Submit your file to Canvas.
    a. The submission must take place before the end of the lab. Canvas will lock the submission after your lab time. If you are switching your lab day (perfectly fine), you must tell us before your assigned day.