Solution Design: Data Structures

TrafficData

A struct containing the traffic data retrieved from the input file.
Since each data point contains a timestamp, a traffic light id, and the number of cars that pass through, the TrafficData struct provides a convenient container for holding each data point andlooks like the following:

```
struct trafficData {
    int timestamp;
    int light_id;
    int cars;

    inline bool operator==(const trafficData& rhs) const;
    int getHour();
};
```

The functions are helper functions which overload the == operator in order to compare two TrafficData structs together and also a getHour function which converts the timestamp into an integer hour 0-23.

In addition, an EMPTY_TRAFFIC value is defined which equates to all values set at -1, and represents an empty TrafficData struct.

Buffer

A class containing my own implementation of a bounded-buffer system.
This class manages the storage of values in the buffer using an internal vector that is limited to a size determined by the constructor parameter “size”.

The buffer is thread-safe and uses a mutex and conditional variable to ensure exclusive access to class members that are shared by threads. In this way, no thread can modify a class member without first gaining access to the mutex lock, which can only be held by a single thread at a time.

Short explanations of the private and public members can be seen in the class declaration below:

```
class Buffer
{
private:
    int _size;                      // The number of elements the buffer can store at one time
    std::vector<TrafficData> _values;   // A vector of TrafficData objects containing the elements within the buffer
    int _emptySlots;                // The number of empty slots (represented by EMPTY_TRAFFIC)

    int next_in;                    // The next slot available for production
    int next_out;                   // The next slot ready to be consumed

    bool endOfData;                 // Signals that there is no more data to accept into the buffer

    std::mutex mtx;                 // The mutex variable
    std::condition_variable cv;     // The conditional variable that handles automatic locks and unlocks during wait times

    void push_next(TrafficData data);   // Adds a TrafficData object into the active buffer region
    TrafficData pop_next();             // Removes a TrafficData object from the end of the active buffer region

    void calculateEmptySlots();         // Calculates the amount of empty slots updating _emptySlots so that isEmpty() and isFull() are O(1) operations

public:
    Buffer(int size);               // Constructor taking a buffer size integer
    ~Buffer();                      // Destructor

    void produce(TrafficData data);     // Produces a buffer slot
    TrafficData consume();              // Consumes a buffer slot and returns its value

    bool isEmpty();                 // Returns true when the buffer is empty (all EMPTY_TRAFFIC slots)
    bool isFull();                  // Returns true when the buffer is full (no EMPTY_TRAFFIC slots)

    void exit();                    // Signals that there is no more data to accept into the buffer and notifies via the conditional variable
    bool end();                     // Returns endOfData
};
```

TrafficCongestion

A struct that contains the consumed congestion data from the input file.

After each TrafficData object is consumed from the buffer by a consumer thread, the consumer thread then processes the data by inserting it into the correct index of the TrafficCongestion data map. This map is then accessed after all data points have been produced and consumed to create an hourly report indicating the top N most congested traffic lights.

There are some helper functions which make calculating sums and averages per hour and per traffic light easier, implemented in congestion.cpp.

```
struct TrafficCongestion {
    // Map containing the congestion data for each hour in the format [hour => [light_id => [measurement1, measurement2, measurement3, ...], [light_id => ....], ...]
    // Example structure:
    // [6 =>    [1 => [832,400,244,...],
    //          [2 => [722,100,942,...],
    //          [...],
    // [7 =>    [1 => [832,400,244,...],
    //          [2 => [722,100,942,...],
    //          [...],
    // [...],
    std::map<int, std::map<int, std::vector<TrafficData>>> data;

    int sum(int hour);                                  // Calcualtes the sum of all traffic within a given hour across all traffic lights
    int sum(int hour, int light_id);                    // Calulates the sum of all traffic within a given hour across a given traffic light id
    int avg(int hour);                                  // Calculates the average congestion of all traffic lights in a given hour
    std::vector<std::pair<int, int>> getTotals(int hour);   // Retrieves a vector containing a list of car totals for a given hour for each traffic light
};
```

Solution Design: Design Choices

Input file

As explained by the instruction document, each traffic signal data point required a timestamp, traffic light id, and number of cars passed by value. in my input file, this is ordered like below where each separate data value is delimited by a space, and each separate data point is separate by a newline:

timestamp1 trafficlight1 cars1
timestamp2 trafficlight2 cars2
…

Setting up the input file in such a way allows me to make easy use of the C++ std::getline(std::ifstream, std::string) function and simplifies things later on when implementing producer threads.

In addition, given that each data point contains a timestamp, traffic light id, and number of cars, it makes sense to create a TrafficData container containing these values together.

Flexible threading implementation

In order to implement threading in the first place, I needed to research mutexes and conditional variables, as well as how the producer-consumer pattern is designed. In the producer-consumer pattern, it is important that the following conditions are enforced:

- Producers must block production if the buffer is full
- Consumers must block consuming if the buffer is empty
- The pointer to the next available slot must be accessed and updated by a single thread at a time to avoid two producers overwriting an existing slot
- The pointer to the next available spot to be consumed must be accessed and updated by a single thread at a time to avoid two consumers reading the same slot

In general, to enforce the above rules it is clear that a producer must wait to acquire a lock before writing to the buffer, only when it is not full, and a consumer must wait to acquire a lock before reading from the buffer, only when it is not empty.

This is exactly the implementation I went for, writing a buffer::produce() function that takes a TrafficData struct, attempts to acquire a lock for the critical section and if the buffer is not full, it inserts the data into the next available slot, unlocking the critical section again. Similarly, for the consumer I wrote a buffer::consume() function that attempts to acquire a lock for the critical section, and if the buffer is not empty it will read the data from the slot at next_in index, unlock the critical section and return the consumed data back to from the function.

In this way, it is possible to use a flexible amount of threads given that the input file can be read with any number of threads since there is no writing being performed, and produce() and consume() are thread-safe also.

Processing data

For simplicity, the consumers process data by adding the data point to a map contained within a TrafficCongestion object. This can easily be changed to process the data however the user requires, such as updating a top N list as the program is being run with mutex and conditional variables in a similar way to the produce() and consume() functions (although the program runs exceptionally fast without sleeps so this would be difficult to demonstrate in this scenario). Instead, after the data is produced and consumed, a post-processing step is completed in the main function which reports on the traffic data in the TrafficCongestion data map.

Buffer exit problem-solving

If you recall, one of the conditions of the producer-consumer pattern is that the consumer will wait when the buffer is empty. Because of this rule, when the producer is finished producing all of the data into the buffer, the consumer threads will infinitely wait, halting execution of the program.

To combat this, when the producer threads are synchronized (break out of loop when the ifstream::eof() is reached), Buffer::exit() is called from the main function, which sets the endOfData bool to true.

An extra condition is added to the consume() function to instruct the thread to wait, which is now if the buffer is empty AND if endOfData == false (or just end() == false). If any of these conditions are not satisfied, the thread will stop waiting and continue execution of the consume() function. Normally, this results in a TrafficData slot being consumed from the buffer. However, an extra check is used directly after the function resumes execution which checks if the buffer is empty AND end() == true, this signals that there will be no more data to process, and the function returns EMPTY_TRAFFIC, which is interpreted by the thread function as the time to break from the while (true) loop and exit the thread execution.

An important aspect of this is that in the Buffer::exit() function, cv.notify_all() is called directly after the endOfData bool is set to true. This causes threads that may have already entered the consume() function when the buffer was empty, but **before** the exit() function was called, to have the endOfData bool updated, and hence break out of the wait while loop.

Macros

Several macros are defined in the main function which alter the operation of the program, including the BUFFER_SIZE, PRODUCER_THREADS, CONSUMER_THREADS, NUM_TRAFFIC_LIGHTS, TOP_CONGESTED, and more. Altering these values can demonstrate how the buffer operates under

a different number of consumer and producer threads, as well as increased or decreased buffer sizes. There is also a commented out section of code at the beginning of the main function which allows you to generate your own random data file using some of the predefined macro values.