FINAL REPORT

A review of the literature research on lock-free programming and the architecture and implementation specifics of a lock-free fire dispatch system.

Ethan McNamara
ethan.s.mcnamara@gmail.com
GitHub: ethan-mcnamara

Table of Contents

1 Introduction

At the University of Victoria, a directed study course is a full-credit which provides a student an opportunity to perform further research into a topic of their own interest with support provided by a professor whose research focus aligns somewhat with the research topic. The result of this directed study may be a scholarly paper discussing the research performed or a project built using the tools into which the research was performed.

In this instance, the topic chosen was one mentioned in a previous course taught by the professor, Dr. Sean Chester: lock-free programming. Lock-free programming was introduced in Dr. Chester's Summer 2022 parallelism course "Data Management on Modern Computer Architectures". While the topic was discussed, the course focused on a large breadth of topics and therefore was unable to provide the depth to which the author was interested.

Thus, the topic of lock-free programming was decided upon for the directed study. During the Fall 2022 semester in which this course was administered, the author was working at a co-op job placement at Zetron, Inc. working on the MAX Dispatch product. The MAX Dispatch system is an emergency dispatch system providing an interface by which an operator could assign emergency response crews, primarily fire and police, to situations. This manual mission-critical system was one in which Dr. Chester saw an opportunity for automation and therefore the course project was decided to be an automated fire dispatch system.

1.1 Objective

The primary goal of this project was to perform research into the topic of lock-free programming that exceeded the information provided in Dr. Chester's previous course while also developing a project built using the information acquired during the research.

The deliverables required for this course include a discussion of the academic sources reviewed, a design document for the program, and a final presentation and report summarizing the cumulative work performed. While the program itself is not a deliverable, its successful implementation is an implicit requirement of the final presentation and report.

2 Background

Computer-aided dispatch (CAD) systems provide a valuable interface by which 911 call takers, dispatch operators, and first responders can communicate in a time-sensitive manner.

2.1 Emergency Dispatch Systems

Most modern regional fire, police, and EMS districts accept emergency situation details from a 911 call centre and use a desktop application to pick vehicles to respond to the situation and communicate to the crew the situation details. While this system can work very effectively with smaller districts where the volume of events and vehicles is minimal, with larger districts such as major urban centres, this manual process can be slower than ideal and may require constant communication of some sort between large numbers of people.

2.1.1 Commercial Solutions

Most commercial dispatch systems provide a graphical user interface by which the operator can receive information via a CAD API from the 911 call taker and then assign crews to the event. These systems communicate with the first responders using pagers, radios, and over IP. All widely used commercial products rely on a human operator to determine which vehicles are assigned to a particular event. While this system can work very effectively with smaller districts where the volume of events and vehicles is minimal, with larger districts such as major urban centres, this manual process can be slower than ideal and may require constant communication between large numbers of people.

2.1.2 System Requirements

Due to a dispatch system's requirement to perform real-time scheduling for critical situations, performance and fault tolerance are the system's two most important quality attributes.

2.1.2.1 Performance

Understandably, the sooner first responders can arrive on-scene, the higher the likelihood of a positive result for the victims. As a result, avoidable delays in processing or rending the interface can not occur. In a manual dispatching system, the bottleneck is likely the time taken by the operator to decide which vehicles are to be dispatched, but an automated system can avoid this bottleneck and will instead focus on optimizing the processing speed.

2.1.2.2 Fault-Tolerance

Naturally, reliability and resistance to faults is a crucial component of any system handling life-and-death data. Due to these systems acting as the primary interface through which the operator communicates with the first responders, should the system fail that communication link may no longer be available, causing potentially lethal consequences as those responders may be unaware of a situation requiring their response.

2.2 Dispatch System Simulation

An automated dispatch system, while abstracting many of the real-world details of a true dispatch system, provides much of the same functionality as a commercial solution. This implementation is designed for a fire district, although that is only true due to the naming of the classes as the foundation of the system could be readily adapted to other varieties of emergency dispatching. To improve performance and fault tolerance of the system, a multithreaded programming model is used, with lock-free techniques used to perform the synchronization. Later sections will explain this synchronization in further detail, in addition to providing a compelling rationale for such a design decision.

2.2.1 Simulating Real-Time Events

The focus of this program is the dispatching of the events and not the generation of accurate emergency events in real-time. Therefore, the sample data for the emergency events will be static and provided at run-time in the form of a comma-separated value (csv) file. These events will be loaded at runtime into a vector representing a queue of events ordered by start time. As the start time of each event is reached, it will be moved off the pending queue to an active queue, representative of the information for this event having been received from the 911 call taker.

2.2.2 Automation Concerns

The primary reason why commercial dispatch systems do not use automated dispatching is that the performance benefit is often outweighed by the lack of trust in an accountable operator. Dispatch operators are trained extensively in the use of the system and can reliably dispatch crews effectively, but should a mistake occur, they are ultimately accountable. Given a typical stream of events, the increased performance provided by an automated system is not often required. However, in a situation where the volume of events is in sufficient excess, an automated system can provide performance not available through traditional operators.

2.2.3 Multithreading

The ability to dispatch crews to multiple events simultaneously is the real benefit of an automated dispatch system and therefore a multithreaded approach is the ideal manner in which to develop such a system. In addition to increasing the performance of the system, the use of multithreading also increases the robustness of the system as the failure of an individual thread may cause an issue with a particular event but will not affect the system’s ability to dispatch the other events.

2.3 Lock-Free Programming

To improve the performance and fault tolerance of the automated system, a lock-free algorithm is used to determine which vehicles are dispatched to each event. Lock-free algorithms are those algorithms such that at all times at least one thread is making progress. An algorithm is considered lock-free if infinitely often a thread will complete its task in a finite number of steps.

2.3.1 Lock-Free Design Patterns

Lock-free programming, true to its name, does not use locks in its implementations. Instead, atomic instructions (instructions that are guaranteed to occur sequentially without any interruption by other instructions) are used to modify the shared data structures. Atomic instructions are used as they allow threads to modify the data at a single exact instance, without the possibility for another thread to modify the shared data at the same time.

There are sets of atomic instructions used commonly in lock-free algorithms. The availability of each instruction set depends on the manufacturer of the CPU on which the program is run. The compare-and-swap (CAS) instruction is the more popular of the two, with some machines instead implementing the load-linked (LL) and store-conditional (SC) instructions. The two instruction sets offer different functionality but can be used to accomplish similar tasks, with only slight modifications needed to adapt a program from one instruction set to the other. For this project, the CAS instruction was used as the hardware on which the system will run does no support the LL and SC operations.

2.3.1.1 CAS Instruction

The CAS instruction compares a provided value with the value stored in a provided memory address. If these two values match, the third parameter is assigned to the memory address. If the two values do not match, the value in the memory address is assigned to the local variable, hence the swap implied in the instruction’s name.

The common implementation will involve a do-while loop where inside the do statement the value is read and stored in a temporary variable before writing to another temporary variable the desired new value of the variable. Inside the while statement a CAS function is called that checks whether the value read and stored is still equal to the variable in memory in which case the new desired value is written and breaks from the loop and if not, the loop reiterates.

2.3.2 Proving Correctness of Lock-Free Algorithms

For an algorithm to be correct, it must have safety and liveness. [1] Safety is the guarantee that "bad things" will not happen and defines liveness is the guarantee that "good things" will eventually happen. Individually, these requirements can be trivially satisfied but together prove an algorithm to be correct.

When considering lock-free algorithms, the property that must be proven to prove correctness is the property of linearizability.

2.3.2.1 Linearizability

Linearizability is a property which if proven, describes an algorithm whose list of operations can be described in a linear or sequential manner. The execution history of an algorithm (the list of every operation an algorithm executes, which for a multi-threaded algorithm may not be a sequential list) is linearizable if it is equivalent (containing the same invocation and return operations with the same argument values) to a sequential execution that respects: (a) object semantics, and (b) "real-time" order. [1] Object semantics refers to the fact that the requirements of the call are met, i.e. a pop operation will always return the top of a stack. "Real-time" order is the property that the return values of operations will reflect the true order in which they were called, i.e. two subsequent pop operations will properly return the initial head value to the first call and the next head value to the second call.

An equivalent explanation of the linearizability property is that every operation must appear to happen instantaneously at a single moment in time. For simpler algorithms, this may be a line of code in which an atomic operation was invoked. For more complex operations, this may not be so trivial. This point in the execution is known as the linearization point. The linearization point is frequently used when proving an algorithm's linearizability.

A common mistake in proving an algorithm's linearizability is believing an operation's linearization point is after it completes its required task. Often, the linearization point is located after an operation has completed a portion of its task but still has additional work to do. This additional "clean-up work" could be done by any thread, not necessarily the thread who performed the work preceding and including the linearization point. Since the linearization point is the point at which the task has instantaneously occurred, the algorithm could be designed that between that point and the true end of the task, other threads may recognize that the task is not fully complete and complete the task. Therefore, when proving the linearizability of an algorithm, it is important to recognize where the dividing line exists between the work including and preceding the linearization point and the "clean-up work". Incorrectly recognizing this point could render an incorrect or invalid proof of linearizability.

3 Program Architecture

This project will be developed in C++ and its design is built according to the features and limitations of the language.

3.1 Overview

Leveraging the C++ language’s object-oriented feature set, the program will be built using classes and objects. The procedural (C-style) programming features available in C++ will also be utilized. Specifically, pointers will be used to reduce the cache size of lists of objects when the contents of the objects are not consistently accessed.

3.2 Class Structure

Following object-oriented design principles, the program is divided into multiple classes each containing member data variable and the relevant functions.

3.2.1 Vehicle Class

The Vehicle class represents an entity that carries crew members and water from a fire station to an emergency. The abstract Vehicle class will be extended by the FireEngine and FireLadder classes. These non-abstract classes will retain all the attributes of the abstract class but will carry different equipment and will thus be fit for different critical situations as needed.

3.2.1.1 Vehicle Polymorphism

The FireEngine and FireLadder classes extend the Vehicle class. This polymorphism allows for a single store of all available vehicles without the need to discriminate based on the type of vehicle. This polymorphism was included in the design to better approximate a true dispatch system as there exists several varieties of fire response vehicles each of whom offer different functionality.

3.2.2 FireStation Class

The FireStation class represents a physical entity in which vehicles are stored. This will be the location at which vehicles are located when they are not responding to a call. Each station will be assigned vehicles and crew members, which may change throughout the duration of the simulation as the dispatch system feels is required.

3.2.3 Event Class

The Event class represents an event to which one or more vehicles must respond. Events are timestamped such that they can be simulated to occur live at a particular moment in time. Events also carry vehicular requirements to simulate the differences between true emergencies. These requirements include the number of vehicles required, the number of crew members per vehicle, and the volume of water each vehicle needs to transport.

3.2.4 BitArray Class

The BitArray class holds the 64-bit array representing the variable on which the lock-free synchronization will occur. Each bit represents the Boolean availability status of a vehicle with bit N representing the availability of vehicle N. When the bit is set to 1, the vehicle is currently responding to an event and cannot be dispatched elsewhere while the reverse is true when the bit is set to 0.

3.2.5 Auxiliary Classes

There are additional classes relevant to the program’s execution. The Time class represents a clock with a millisecond granularity. This class is used to determine when an event shall be moved from pending to active. The Location class represents a two-dimensional coordinate on a grid. Travel time is take into consideration when dispatching vehicles and the distance between the vehicle and the event is vital in determining whether a vehicle should be dispatched to an event. The DistrictResources class acts as a repository of the available vehicles and stations in the simulated fire district. The EventFactory class acts as store for the pending and active event queues and the functionality for dispatching crews to an event.

3.2.6 UML Diagram

The below UML diagram in Figure 1 shows the program's classes and their respective member variables.

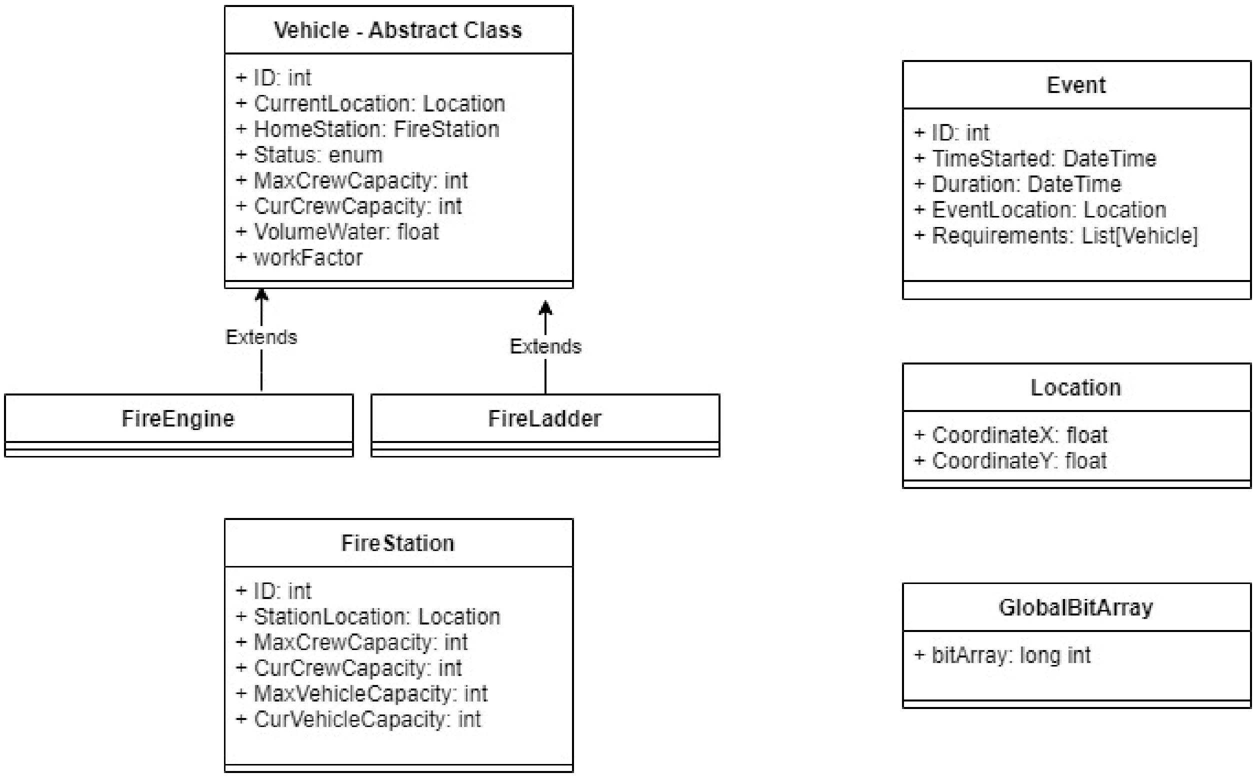

Figure 1: *UML Diagram*

3.3 Data Management

In developing multithreaded programs, the management of data is vitally important to avoid the development of race conditions. To do so properly, an explicit definition of the shared mutable must be made to show the data which may be edited concurrently by separate threads so the proper safeguards may be applied.

3.3.1 Data Flow Design

Recognizing the importance of data management in a multithreaded program, the design of the flow of data between classes is vital and as such models have been created to show the processes involved in moving data at two different granularities.

3.3.1.1 Data Flow Diagram (High-Level)

Figure 2 below details the flow of information between processes from a broad, high-level perspective.

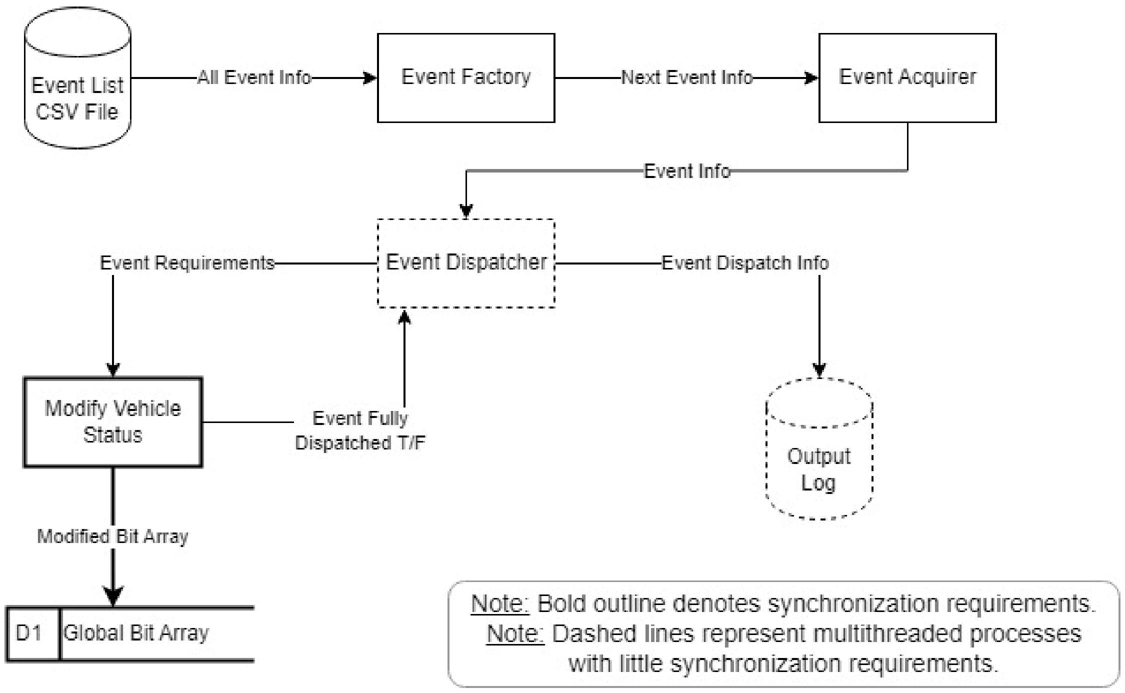

Figure 2: *High-level Data Flow Diagram*

3.3.1.2 Data Flow Diagram (Low-Level)

Providing a higher level of granularity, Figure 3 shows a lower-level DFD.

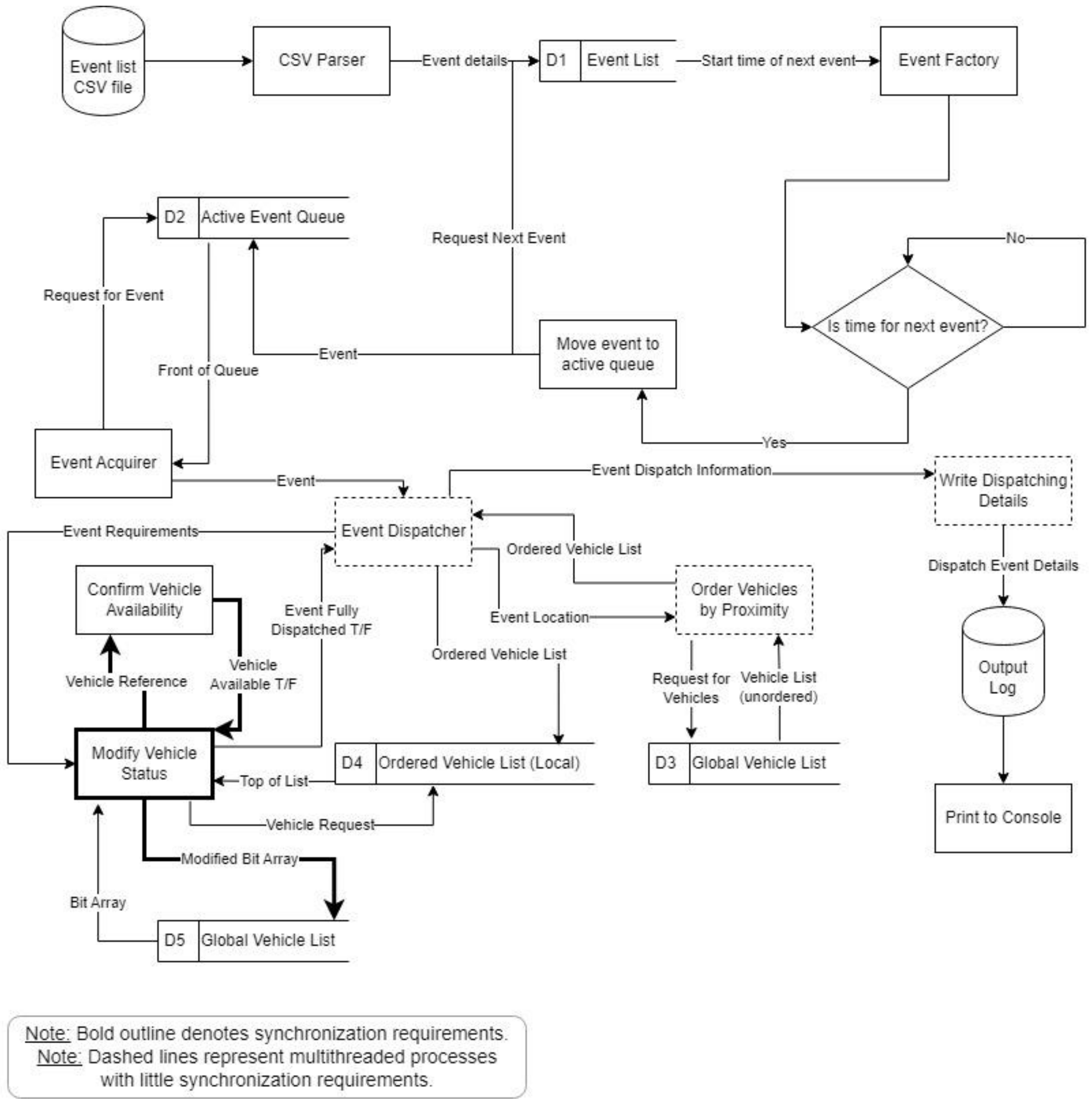

Figure 3: *Lower-level Data Flow Diagram*

3.4 Multithreaded Synchronization

As this project is an effort in lock-free programming, no locks or blocking synchronization methods are used. Nonblocking CAS loops will be used to ensure proper synchronization related to the shared mutable data.

3.4.1 Globally Mutable Data

The data element regularly modified by separate worker threads is the global bit array. This bit array carries the status of whether a vehicle can be dispatched to an event or whether it is unavailable at the current time t be dispatched. In dispatching vehicles to an event, this bit array will be read and a determination will be made for each vehicle whether or not it is available. After a vehicle has completed its response to an event, the bit array will again be modified representing this fact. It is the writing of this data variable around which the lock-free synchronization safeguards are required.

3.4.2 Thread Control Scheme

Apart from the processing of input data, all threads created follow the controller/worker thread structure. Worker threads will be created by the controller thread and then subsequentially detached so that the worker threads will continue to execute without requiring the controller thread to delay until the worker threads have completed. Each event will be handled by a separate worker thread created by the EventFactory, which will act as the controller thread. Within an event, once a set of vehicles are dispatched to the event, each member of the set will be handled by a different worker thread responsible for moving the vehicle to the event, waiting the pre-determined amount of time for the vehicle to respond to the event, and then returning the vehicle back to its home station.

4 Program Implementation

Developed in C++, the program was developed using several standard libraries, including the std::atomic and std::thread libraries to accomplish the lock-free multithreading. Additionally, the sample data was created in a comma-separated value (csv) file format.

4.1 Thread Synchronization

In C++, when creating a new worker thread using the std::thread library, one can either call join or detach on said thread. A call to join requires the controller thread to stop execution until the thread returns while the detach function allows the controller thread to continue its execution without requiring the worker thread to complete its execution. Apart from the threads processing the input sample data, the detach function is called on all other worker threads created in this system.

A single worker thread is responsible for moving events from the pending queue to the active queue. This thread also creates a new worker thread for each event responsible for dispatching crews to this event. This thread is quickly detached so the handling of any single event does not affect the movement of events from the pending to active queues. This event worker thread then follows the program's algorithm to determine a list of vehicles which the event wishes to have dispatched to it.

Provided the list of vehicles selected by the worker thread were dispatched to the event, which may not occur if another event thread has already claimed one or more of those vehicles, the event thread then creates a new worker thread for each vehicle in the list. The vehicle worker threads are responsible for moving the location of the vehicle from its current location to the event location. Once at the event location, this worker thread spins for the event's pre-determined duration before setting the global bit array to represent the vehicle's new availability, before moving its location back to the vehicle's home station. If during the transportation back the station the vehicle is dispatched to another event, the worker thread moving the vehicle will immediately return as the responsibility for the vehicle is transferred to a thread created by the new event.

Event worker threads return after creating worker threads for dispatched vehicle. Vehicle threads return after the vehicle returns to the home station after responding to an event or the vehicle has been dispatched elsewhere. The event factory thread, the one responsible for moving events from the pending queue to active queue, returns after the pending queue is empty and all events on the active queue have been assigned to worker threads.

4.2 Global Bit Array

The global bit array is the shared data on which the CAS function is called. The 64-bit unsigned atomic integer is a public member variable of the GlobalBitArray class. This class contains functionality for returning modified copies of the class variable based on the provided vehicle ID value.

4.2.1 Program Linearization Point

The modification of the global bit array occurs by the worker thread created to handle a particular event. This worker thread obtains a list of vehicles it desires and use the IDs of those vehicles to obtain a modified copy of the global bit array. Then, using the CAS function, the modified copy of the bit array is written to the global bit array. The successful execution of the CAS function is the program's linearization point as it is at this point the vehicles in the list are considered dispatch to that event and cannot be dispatched elsewhere until the event has been managed.

5 Reflections

The design and development of this project were enlightening experiences that provided an opportunity to design a complex C++ project largely independently for the first time. While I have developed software in professional and academic settings, including some projects requiring a larger code base, this project was the first of this scale I built from the ground up without the assistance of other developers. This experience has been valuable as I also gained experience in formal algorithm design, which was not a process I have undergone outside of a course.

5.1 Prototype Development

While some basic infrastructure code was written prior to the development of the system design document, much of the code was written after this document allowing the document to act as a prototype in which kinks in the system could be ironed out before development. This iterative design process significantly reduced development time, as the algorithm was already outlined in pseudocode when it came time to implementation. However, the development of the prototype algorithm was an involved process as multiple iterations were required to ensure it rendered a correct solution.

The initial algorithm, which appeared correct from the outset, was shown by Dr. Chester to in fact not be linearizable and therefore could not be proven to be a correct lock-free algorithm. This draft of the algorithm did not use a bit array and instead used the CAS function to modify the vehicle statuses directly. The flaw in this approach was the requirement to modify the status of each vehicle individually, which allowed for the possibility of deadlocks. As a result, the BitArray class was added to act as a single data element whose modification would affect the availability of all vehicles. This limited the maximum number of vehicles available to 64 or 128, depending on whether a double-word CAS function is available on the test machine, but was a necessary compromise to ensure correct program execution.

5.2 Comparison with Previous Concurrency Course

Dr. Chester's previous parallelism course, Data Management on Modern Computer Architecture, provided an insight into lock-free programming, but the further research performed in relation to this course allowed for a greater understanding of the topic. I had understood the practise of lock-free programming and the CAS function but had been unaware of the LL & SC functions which provide an alternative to CAS.

Additionally, Dr. Chester's previous course did not touch on proving the correctness of lock-free algorithms. The concept of linearization was a new one and it provided a new manner of thinking around multithreaded algorithms. While linearization itself is not a necessarily simple concept to understand, the idea that every interaction with a shared data element in a multithreaded system should be able to be ordered in a single sequential history is one that is easily understood and provides an easy way of understanding how multithreaded algorithms should be developed.

5.3 Different Approaches on Project Redo

If provided the opportunity to redo the same project, better incorporating the limitations of lock-free programming would be the biggest change made. While the initial design of modifying a vehicle's status attribute as a means of an event acquiring that vehicle makes sense from a single-threaded approach, it is not effective from a lock-free multithreaded approach. Having a better understanding the restrictions imposed on a program by the CAS function would better inform the project design and result in a shorter design period and easier implementation.

6 References

[1] Michael Scott. "Michael Scott — Nonblocking data structures. Part 1.," YouTube, Oct 14, 2019. [Video file]. Available: https://youtu.be/9XAx279s7gs. [Accessed: Oct 23, 2022].