16.35 - Real-Time Systems and Software
Asst. #3

Handed Out: April 8th, 2020
Predeliverable Due: April 23th, 2020 at 5:00 PM Boston
Due: May 7th, 2020 at 5:00 PM Boston

Submission Instructions

Please submit your assignments electronically using `git`, and upload the files to Gradescope. Some notes about the submission:

1. Please work within the code we have provided on learning modules. Do not try to build off your previous pset solution.
2. The written section will require you to report the time that you spent on each question. Please bear this in mind when developing your solutions.
3. There are two Gradescope assignments that you will have access to: one for the predeliverable and one for the final deliverable. Make sure you upload to the correct assignment. Because we are asking you to edit header files, there is no autograder set up for this assignment. Please directly upload the whole assignment folder to Gradescope (the folder including `include`, `src`, etc.).
4. This project is collaborative! Find a partner and a cool team name on this google sheet. (Full URL in case the clickable text doesn't work: https://bit.ly/2V4drBq). We're an odd number of people, so one team can have three members or someone can work alone. Only one member of the team (Member 1 in the sheet) must upload the final submission.

Please ensure that your code compiles and runs from within your source folder. You will not receive grades for code that does not compile.

Setup

This code, as provided, supports UNIX operating systems. Your assignments from here will make extensive use of the `pthread` package, known as POSIX-threads. POSIX-threads are not (natively) supported on Windows. We highly advise setting up and working with a virtual machine and/or a linux distribution for this and future assignments.

You are welcome to modify the code provided to work on Windows but note that no support will be given.

The code provided should not require the installation of additional packages (although a barebones installation may require you to install `cmake` and `python2.7`).

Compiling

The project provided will contain a file called `CMakeLists.txt`. `cmake` is a build system - it generates the makefiles that one would traditionally write, and provides high-level functionality to intelligently do so. `cmake` is extremely widely used; it is simple, exact and has a nice testing framework that we'll find useful in this assignment. Note that other build systems exist and have comparable capabilities - eg. gradle, autoconf, Premake, Ninja etc.

Please follow these steps for building the project:

1. Navigate into the directory containing the assignment files (you should see a file called `CMakeLists.txt`).
2. Create the build folder and change directory to it
   `$ mkdir build && cd build/`
3. Now run `cmake`, with the root folder in the parent directory:
   `$ cmake ..`
4. `cmake` will generate the make file for you. To build the application, simply run
   `$ make main`
5. To run the application, simply call `./main` in the build directory:
   `$ ./main`

Assignment Overview

This assignment will require you to implement a fully threaded solution based loosely on the later parts of the previous assignment. We will provide you with a working solution for the previous assignment. Be warned that the code has been slightly modified from the specification required for the previous assignment - this is to make the later parts of this assignment easier for you to implement, and to make the entire process more amenable to threading. You are welcome to integrate your previous solution as you feel is appropriate, should it make any part of the provided code more clear to you. We are primarily interested in approaches to threading for this assignment.

The predeliverable will require you to implement a more appropriately threaded version of the previous assignment. The vehicle will make use of the waypoint control method we developed previously but each vehicle will run on a separate thread. The simulator will execute from the main thread. You will need to implement the appropriate synchronization functionality using condition variables and to protect the shared critical region using a mutex.

The final part of this assignment will require you to implement several different (but simple) controllers that will introduce new shared critical regions that you will need to guard appropriately, followed by a short written section.

Partial credit will be given for incorrect implementations, provided that they **compile** (code that does not compile will receive no credit). The total number of available points is 230.

1 Threading and Synchronization [Predeliverable - 55 Points]

In this section we will explore a simple approach to threading. Please use the provided code as a base to work from. The code is largely identical to the code you developed for your previous assignment, but has structural changes more appropriate for the later sections of this assignment.

In this design, each vehicle runs on its own thread, while the simulator runs on the main thread. The simulator is responsible for incrementing time, but must wait until all the vehicles have updated. The vehicle is required to update in its own thread, but after updating must wait until all other vehicles have updated and the time incremented by the simulator before updating again.

You need only use condition variables and mutexes to enforce this behavior. We have provided you with working vehicle, simulator, and controller classes. Your job will be to modify the provided code according to the following rules:

1. Create N threads in the `main.c` file, where N is the number of vehicles. Each vehicle will get its own thread. The simulator will run on the main thread that starts when you run your program.

2. The vehicle class has been augmented with a `run()` method. This method is currently blank but will need to be called when the thread is initialized. You will need to implement this method to

   - Control the vehicle (i.e. call the `void control_vehicle(struct t_vehicle * v)` method)
   - Update the vehicle state (i.e. call the `void update_state(struct t_vehicle *v, double time)` method)
   - Increment the simulator's counter variable (`int vehicles_updated`) that keeps track of how many vehicles have been updated. Remember that this will run concurrently on multiple threads. You will need to protect this region with a mutex.
   - Notify the simulator that it should check the tic condition (have all vehicles finished?)
   - Wait on the simulator's condition variable before reverting to the first point (i.e. all this in a loop).

   The vehicle's run method shall terminate when the simulation current time exceeds the maximum time.

3. The simulator's run method will need to be augmented with appropriate synchronization mechanisms to ensure that:

   - The simulator thread waits until all vehicles have updated (i.e. the simulator should check the condition that `vehicles_updated == n_vehicles`) **before** incrementing the time.
   - After incrementing the time the simulator should broadcast to all waiting threads (i.e. the vehicles) that they may continue. Be sure to use the `pthread_cond_broadcast` method instead of the `pthread_cond_signal` method (otherwise you will only wake one of the waiting threads).

   We have provided you with skeleton code for this method and commented the places where you will need to add the relevant synchronization and locking mechanisms.

4. Please ensure that all pthread structures and types are appropriately initialized and deallocated.

2 Analyzing Critical Regions [Predeliverable - 35 points]

In the implementation you wrote for question 1, we asked you to protect critical regions of the code that could cause issues if you didn't surround them with condition variables. In this section, you will analyze some of the decisions we made for you. Please respond to the following questions in English (as opposed to just pasting code), but do refer to specific fields and operations from the code in your answers.

1. Why is waiting on a condition variable better than a busy-wait loop?
2. Why is no synchronization or protection required for the `control_vehicle` and `update_state` method calls? In the next section we'll explore cases where we will need protection and/or synchronization here. Please give a description of a controller that would have such requirements (examples from the following questions will receive zero credit).
3. What is the critical region in this program? Please explain.
4. Why do we need to protect the incrementing of the `vehicles_updated` variable? Give a concrete example of what could happen if this was not protected.
5. Is a mutex appropriate for incrementing the `vehicles_updated` variable? Please explain. You may contrast the use of mutexes in this case to semaphores and a reader-writer locks.
6. Is there a possibility of deadlock or synchronization failure in this design? Please explain.

3 Follow the Leader [Final - 20 Points]

In this section, you will finish a partial implementation of a new controller called the follower controller. The follower controller, when fully implemented, will allow a vehicle to follow another, "leader," vehicle around. Inside `controller.h`, you'll see three sections that are not currently being used: a new struct called `follower_control_data` as well as two new methods. You will use the structure and implement the methods for this problem.

3.1 Structure Overview

The three components that you just reviewed work together in an extremely powerful way. The `get_follower_control` declaration says that, when given a vehicle `v`, this method will return a control object that will direct `v` towards its "leader." Because `get_follower_control` returns a `control` struct, it can fit in seamlessly with the rest of the code.

How, though, will the `get_follower_control` know what leader for a vehicle to follow? That's where `follower_control_data` comes in. You can imagine that, if we implemented many different types of controllers, we might need to pass around different information for each different controller type: the waypoint controller needs to track waypoints, the follower controller needs to track a leader, an "average" controller might try to go to the middle of all existing vehicles and therefore would need to track them all, etc. Rather than including all that information inside the vehicle struct directly, we've created a new field for vehicle called `control_data` (take a look at `vehicle.h`) that can store arbitrary information for use by different controllers. It is declared as `void*` since we don't know ahead of time what type of controller will be used for the vehicle. We've added a new field called `control_type` that can help with this. It will be set to specify the type of controller being used, which can be useful for knowing how to appropriately cast `control_data` from `void*` to a usable structure and for conditioning on the type of controller being used if needed.

Control methods can then read from the passed-in vehicle's `control_data` field, trust that the control data matches the required format for that specific controller, and thus compute the right control. All we need to do, then, is make sure that when we initialize the vehicle and set its control policy, we also correctly set its control data and specify the type of controller being used. For the follower controller, `create_follower_control_data` can help initialize the control data.

3.2 Implementing the Controller

Your job is to get one of the vehicles in your simulation following another one. To do so, you will need to implement the methods you declared inside `controller.h`. You will also need to update `main.c` to initialize one of the vehicles so that it can use the follower controller. Use the existing code that sets up the waypoint controller for inspiration.

Implementing `get_follower_control` will take special care. First, you must decide on how to implement the behavior of "following" a leader. We recommend drawing inspiration from your proportional waypoint controller. Second, you need to make the follower controller thread-safe. The behavior of the follower controller now depends on values that can be edited by another thread. You can coordinate the threads through a locking mechanism. In order to implement the locking, you will need to edit `vehicle.h`, `vehicle.c`, `controller.c`, and `simulator.c` (simulator only for cleanup).

4 Getting Your Ducks in a Row [Final - 25 Points]

In this section, you will analyze some pseudocode for potential deadlock.

Consider a situation in which all your vehicles in the flight simulator use a follower controller designed to follow the next vehicle (where "next" means the i^{th} vehicle follows the $i+1^{th}$ vehicle, except for the last vehicle, which follows the first vehicle)[1]. Furthermore, imagine that you have implemented a follower controller that obeys the following structure:

[1] https://xkcd.com/537/

```
control get_follower_control(struct t_vehicle * vehicle) {
    lock(vehicle->position_mutex);
    lock(vehicle->control_data->leader_vehicle->position_mutex);
    ...
    // Compute relevant control c to drive vehicle toward the leader.
    ...
    unlock(vehicle->control_data->leader_vehicle->position_mutex);
    unlock(vehicle->position_mutex);
    return c;
}
```

Please answer the following questions about the possibility of deadlock:

1. The above code could lead to deadlock. Please explain how deadlock could occur. Provide a specific example of how deadlock could occur in a scenario with two vehicles and describe what behavior a user would observe.
2. Describe how one could modify the `get_follower_control` implementation to compute the control while simultaneously holding exclusive access to both the current vehicle's position and the leader position's and still prevent deadlock. We will not accept the "ostrich" approach of give up and start over.
3. If we removed the requirement that `get_follower_control` locks on `vehicle.position`, could deadlock arise? Is removing the lock a reasonable thing to do given the current threading behavior of one thread per vehicle?
4. Assuming you have implemented a solution that removes the possibility of deadlock, please analyze the performance of this program if all your vehicles used a follow controller and they all followed the same vehicle. What if they all followed different vehicles?

5 Off on Your Own [Final - 30 Points]

Create your own controller from scratch. At this point, you have two examples of working controllers, so figuring out the wiring for a third one should not be too challenging. This new controller can do anything you'd like as long as 1) you can reasonably argue that it's "cool", 2) it exhibits fundamentally different behavior from the previous two controllers, and 3) it entails the coordination of at least two vehicles.

Implement your controller and write the desired behavior in a big block comment above the implementation in `controller.c`.

If you are truly desperate for inspiration, we suggest an "average" controller that drives a vehicle to the average position of the other vehicles, a "misochime" controller (playing fast and loose with Ancient Greek for hate and $o\chi\eta\mu\alpha$ for vehicle) that flees one particular vehicle or all of them, a "clique" controller that selects a few other vehicles to follow but avoids others, etc. Clearly, you can get really interesting and complex interactions if you implement a few of these controllers and have them run at the same time.

6 Identifying Critical Regions [Final - 25 Points]

In this section, you will analyze the concurrency properties of the controller you wrote in the previous section by answering the following questions:

1. What critical regions of the code does your new controller depend on? Do you need to introduce additional synchronization to protect the regions?
2. Does your new controller introduce the possibility of deadlock? If so, what strategy did you implement to avoid it?

3. What sort of performance would you expect to see if all your vehicles used your new controller? We are specifically asking you to analyze computational performance - i.e. how long would the simulation take to run - rather than behavior. We do not need concrete numbers, but we would like you to point out when your controller will end up waiting for a lock and how performance scales with the number of vehicles.

7 Scheduling [Final - 30 points]

In this section, you will demonstrate your knowledge of scheduling algorithms and properties by analyzing a hypothetical aircraft control scenario. No coding is necessary, just a quick write-up of your answers, with any numerical calculations briefly justified.

Consider the following properties of an aircraft that you would like to control. The aircraft itself is outfitted with an IMU and a GPS, but no pilot! The good news is that you've been put in charge of the project to sensibly poll the aircraft's sensors to keep the plane in the air.

The aircraft control loop operates at 100Hz, and computing the control each time takes 1ms. Reading an attitude estimate from the IMU also operates at 100Hz but takes 2ms. Acquiring a GPS position estimate takes between 10 and 100 ms, but only needs to be done at 10Hz. Please identify the scheduling parameters and constraints present in this problem, and give a scheduling algorithm that will enable you to solve this problem.

8 Report the Time that you Spent on Each Question [Final - 10 Points]

Please report the time spent on Questions 1 - 7.