# Implementing thread management

The core of technology of our application is the use of threads, which allows the simulation to be similar to how a car works, in the sense that the parts of a car are not one large piece, it is made up of individual components that work together, passing instructions or information or even substances to each other for the components to function and realise the feature of the car. Also from the software development standpoint, it allows us to be able to take advantage of the multithreaded nature of modern CPUs in order to improve performance, and to a certain extent, also isolate each subsystem from the other subsystems.

In our application, threads will be used in the following ways:

* Each main subsystem of the application (simulation, infographic, and quiz) should run in their own threads, independent of the main menu screen that allows us to choose which subsystem we would like to go to
* In the simulation subsystem, each component of the car being simulated should be in their own threads, with inter-thread communication allowing us to be able to handle information sharing among the threads.

For the first use of threads, that is the implementation of each subsystem in their own threads, it is not too difficult as all we would need would be to create the thread and call the function that acts as the “starting point” for the subsystem to begin the functionality of that subsystem. As each subsystem is independent of each other, we won’t really need to implement any form of information sharing between the various subsystems in the application.

However in the case of having each component of the car in the simulator to be in their own threads, it is not so easy, as we would need to implement some form of information sharing between the threads. For example the fuel tank needs to send a message to the instrument cluster when the level of fuel is low, and the fuel tank will need to supply fuel to the engine for it to be able to run.

Threads make it a bit easier to implement the sharing of information in the sense that threads are able to access all the variables that have been created in the parent function that creates and starts the thread. Because of this, we would be able to create a set of variables that can be accessed by the threaded functions, and the threaded functions can then grab the information stored in those variables to vary their execution. To prevent race conditions, we could implement some mutual execution locks in order to ensure at any one time only one function accesses the variable. This would be necessary mainly if there is more than 1 function being able to access the variable at the same time, which in the case of our application, would be highly possible. For example, if we have a function running in that calculates the fuel consumption of a car by getting the fuel level stored in a variable in the parent function that created that thread, and at the same time we have fuel tank that regularly updates the value of the fuel level in the same parent function, we would most definitely need a mutex lock as this would prevent instances where the fuel tank updates the value in the variable the same time the fuel consumption function gets the value stored in the variable.

Implementing the mutexes are not that hard however, as all we would need is to use the functions that have been predefined in the pthread library which will allow us to easily setup the mutexes in our application. However, care must be taken during the development process to ensure that once done, the mutex is always unlocked to prevent issues from deadlocks, especially in the cases when an error occurs and if the execution of a block of code between a mutex lock and unlock is aborted before the mutex is unlocked.