Things to do:

#cars

To begin with, create a struct to represent each car. This struct should have the following fields:

- A mutex to protect the square that the car is currently occupying.
- An integer field to represent the current position of the car in the traffic circle.
- A thread ID field to represent the thread that is simulating the movement of this car.

#road

then create a 2D array of these structs to represent the traffic circle, with each element representing a square in the circle.

#car threads

The thread function for each car should do the following:

1. Try to acquire the mutex (1) of the next square that the car wants to move to. If the mutex is already locked, wait until it is available.
2. If the mutex was acquired successfully, move the car to the next square and release the mutex of the current square.
3. Repeat this process until the car reaches a sink. At this point, check if the car should disappear with probability FIN_PROB. If it should, exit the thread. Otherwise, continue moving around the traffic circle.

(1) You can use the pthread_mutex_trylock function to try to acquire a mutex without waiting. If it returns an error, you can print an appropriate message and exit the simulation.

#generating function

Next, create a function called "generate_car" which will be responsible for generating a new car at one of the car generators. This function should do the following:

1. Select a generator randomly (one of the four corners of the traffic circle).

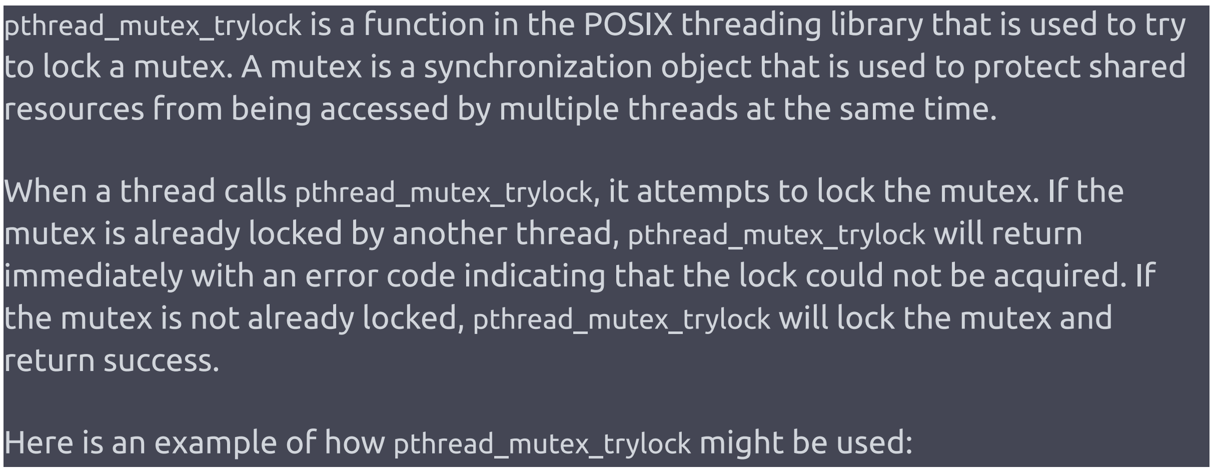
2. Select a random time between MIN_INTER_ARRIVAL_IN_NS and MAX_INTER_ARRIVAL_IN_NS to wait before generating a new car.
3. After the wait, generate a new car struct and create a new thread for it. The thread function should be responsible for simulating the movement of the car around the traffic circle.

#main function

Finally, create a main function that initializes the traffic circle, creates the four car generator threads, and then waits for the simulation to finish before destroying the mutexes and exiting the program. You can use the pthread_join function to wait for a thread to finish.

Remember to minimize the sections of code that require mutual exclusion, as well as to avoid deadlocks. You can also print a snapshot of the traffic circle every so often (e.g. every second) to see how the simulation is progressing.

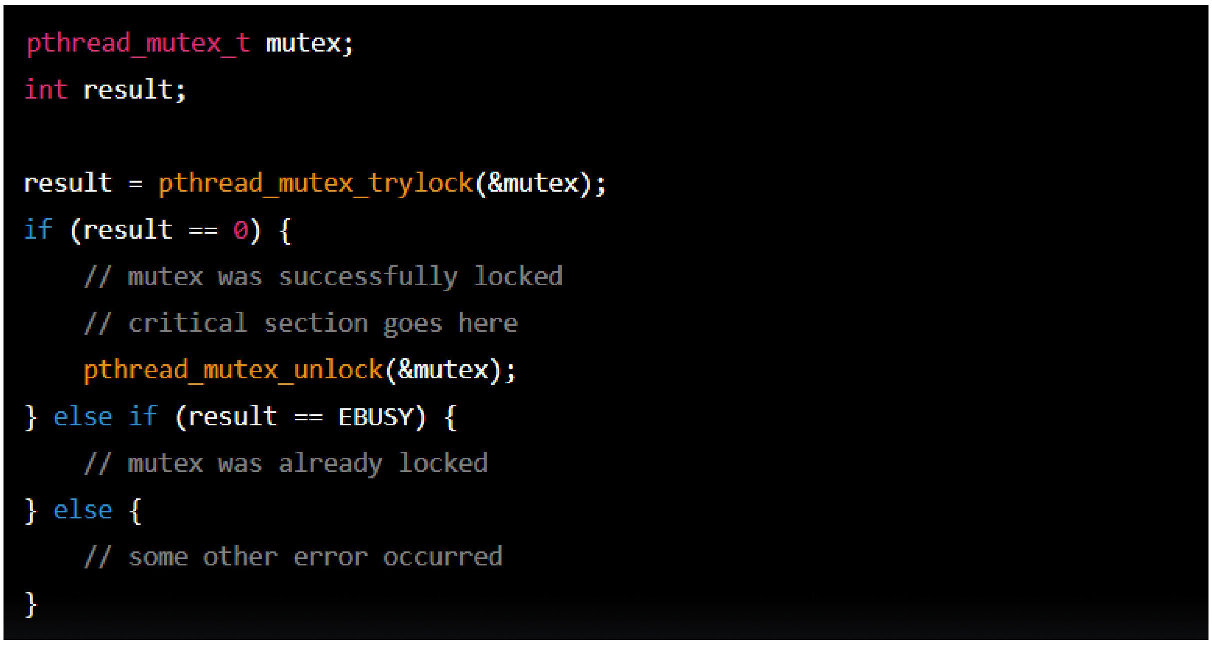
Docs of used functions:



pthread_mutex_trylock is a function in the POSIX threading library that is used to try to lock a mutex. A mutex is a synchronization object that is used to protect shared resources from being accessed by multiple threads at the same time.

When a thread calls pthread_mutex_trylock, it attempts to lock the mutex. If the mutex is already locked by another thread, pthread_mutex_trylock will return immediately with an error code indicating that the lock could not be acquired. If the mutex is not already locked, pthread_mutex_trylock will lock the mutex and return success.

Here is an example of how pthread_mutex_trylock might be used:



```
pthread_mutex_t mutex;
int result;

result = pthread_mutex_trylock(&mutex);
if (result == 0) {
    // mutex was successfully locked
    // critical section goes here
    pthread_mutex_unlock(&mutex);
} else if (result == EBUSY) {
    // mutex was already locked
} else {
    // some other error occurred
}
```