In my code, main() is responsible for taking arguments from the CLI, checking their validity, initialising the barrier (described later), and then creating and keeping track of fans (threads). It uses a couple of simple functions while performing the checks, but the section is overall very straightforward, and the code is easily readable. I used comments generously (like in the remainder of my code), which should help.

The most crucial part of my code is fan\_behaviour(void\* arg), which is the function called by each thread. It contains the critical sections that I protected via a global lock and multiple phases, through which each thread passes under the supervision of a barrier. Its pseudocode is as follows:

1. Cast the argument *void\* arg* provided to the function as a parameter to *fan\* person*.

* Note: *fan* is a struct that holds the name of the team a given fan supports.

1. Lock the mutex (globally defined).
2. Output thread and team information, and print **init**.
3. Increment the waitlist corresponding to the fan’s team by 1 (waitlists are globally defined).
4. Determine whether a rideshare can be formed:
   1. If there are a) at least two A fans and at least two B fans, b) or four A fans, or c) four B fans:
      1. Update the waitlists accordingly:

* Decrement both waitlists by 2 if **a** holds.
* Decrement the waitlist for A by 4 if **b** holds.
* Decrement the waitlist for B by 4 if **c** holds.
  1. Otherwise, pass (move on to the barrier).

1. Unlock the mutex.
2. Wait at the barrier (initialised in main() so that it can hold up to 4 threads - that is, it lets all threads proceed to the next phase only when a rideshare band of 4 is formed).
3. Lock the mutex again.
4. Increment the barrier counter (globally defined as *barrier\_counter*; counts the number of threads that successfully pass through the barrier).
5. Output thread and team information, and print **mid**.
6. If *barrier\_counter* % 4 == 0 (if the fan is the captain in a car): output thread and team information, and print **end** with the specific car ID. Then increment the car ID by 1 (globally defined as *cid*).

* Otherwise, pass.

1. Unlock the mutex.
2. Delete the fan pointer to avoid memory leaks.

I wrote a bash script that checks whether a run was correctly executed using the correctness variables defined in the assignment guide and takes the fan numbers given to ./rideshare as input (correctness\_check.sh). When a faulty run occurs, it returns the variable for which the defined conditions were not met and prints the output so that I can look into it. Since threads randomly run, a single check by this script is of course insufficient, so I wrote another script that repeatedly runs correctness\_check.sh. The final version of my program did not return a single error in hundreds of runs, so I have grounds to feel confident that it is correct (not that it *ensures* that it is correct). I also manually verified the condition about the order of **inits**, **mids**, and **ends** in several outputs.

I avoided race conditions by locking and later unlocking any section that included access to a variable shared by all threads (e.g. incrementing/decrementing waitlists, incrementing the car ID, incrementing *barrier\_counter*). Threads are run (pthread\_create()) one after another in main() and are all waited for (pthread\_join()) so that the program does not terminate before they all complete execution. The barrier is initialised (pthread\_barrier\_init()) again in main() for 4 threads and makes threads block until the barrier holds all 4 threads necessary to form a rideshare band. After that point, it allows the threads to move on to printing **mid** and **end** and eventually completing execution. I chose using a barrier instead of semaphores or condition variables, because I think it fits the bill quite well: separating two phases (namely, 1) waiting for and 2) forming a rideshare band).

**init** is printed once by each fan in the very beginning, bound to no condition -hence, **# of** **inits = numA + numB. mid** is printed once by each fan in a rideshare band. In other words, any thread that passes through the barrier prints it (no other condition is required). Therefore, **# of mids = numA + numB**. Finally, **end** can only be printed once by a rideshare band, as *barrier\_counter* will be a multiple of 4 whenever a band forms and only once - thus, **# of ends** = **(numA + numB) / 4** (the number of rideshare bands). Furthermore, the thread printing the statement is always a part of the band, as the first thread to make *barrier\_counter* % 4 == 0 is the last thread to join a band and therefore the captain of the respective band. By extension, this also ensures that exactly 4 **mids** precede each **end**, as the captain will print **mid** before it prints **end** (combined with **mids** printed by the remaining 3 fans, this makes 4).