Intersection Control System: Traffic Lights and Vehicle Protocol

In the context of urban traffic management, we examine a standard intersection equipped with traffic lights, encompassing four cardinal directions (E, W, S, and N), each serviced by a single lane. The intersection operates under a sophisticated traffic control system, featuring three distinct colors: Green, Yellow, and Red. The prescribed light sequence for each direction is as follows: Green persists for TG seconds, followed by Yellow for TY seconds, and Red for TB seconds. This pattern cyclically repeats, alternating between Green, Yellow, and Red. It is imperative to note that the lights on E and W are synchronized, as are those on S and N. Furthermore, a synchronization protocol dictates that when the lights on E and W display Green or Yellow, the lights on S and N must concurrently display Red, and vice versa. This temporal coordination is established by the requirement TG + TY = TR.
Within this system, the functionality of all traffic lights is consolidated into a single-threaded representation, responsible for defining the colors for all directions. The subsequent discussion outlines the comprehensive traffic control policy designed to regulate vehicle movement within the intersection.

Vehicle Representation and Subroutine Execution

The core entity in this traffic management system is the vehicle, each represented by a thread. Upon reaching the intersection, a vehicle executes a well-defined subroutine sequence:

```
Car(directions dir) {
ArriveIntersection(dir);
CrossIntersection(dir);
ExitIntersection(dir);
}
```

The data type 'directions' is defined as:

```
typedef struct _directions {
char
dir_original;
char
dir_target;
} directions;
```

The attributes 'dir_original' and 'dir_target' denote the original and targeted directions of the vehicle, respectively.

ArriveIntersection Subroutine

When a vehicle approaches the intersection, it undergoes a thorough evaluation based on the presence of other vehicles, the traffic light status, and the specific direction of travel.

- Green Light: If the light is green, the vehicle must adhere to specific rules based on its intended maneuver, ensuring safe and efficient traffic flow.
- Yellow Light: During a yellow light phase, careful considerations are made for right turns to avoid interference with other vehicles.

- Red Light: Vehicles approaching a red light must patiently wait for the subsequent green phase, following a prescribed set of rules based on their intended actions.

It is essential to highlight the incorporation of head-of-line blocking when a vehicle encounters another with the same original direction.

CrossIntersection Subroutine

The CrossIntersection subroutine simulates the vehicle's traversal of the intersection, considering fixed time periods (ΔL, ΔS, and ΔR) for turning left, going straight, and turning right, respectively. Debug messages are generated during this process, and the Spin function is employed to emulate the crossing.

ExitIntersection Subroutine

Upon completing the intersection traversal, the ExitIntersection subroutine is invoked to signal the conclusion of the crossing. This subroutine ensures the facilitation of subsequent vehicles to cross the intersection.

Design and Testing

Explanation:

- The code implements a simulation of cars arriving at an intersection, crossing, and then exiting.
- Each car is represented by a thread (pthread_t), and the simulation involves synchronization using mutexes (pthread_mutex_t) and semaphores (sem_t).
- The carInfo struct holds information about each car, and the startCrossing function is the main function executed by each thread.

Testing:

```
Date and Time: Mon Nov 13 08:44:32 2023
Time 1.0: Car 0 (^ ^) arriving
Date and Time: Mon Nov 13 08:44:32 2023
Time 1.0: Car 0 (^ ^) crossing
Date and Time: Mon Nov 13 08:44:33 2023
Time 1.9: Car 1 (^ ^) arriving
Date and Time: Mon Nov 13 08:44:33 2023
Time 1.9: Car 1 (^ ^) crossing
Date and Time: Mon Nov 13 08:44:34 2023
Time 3.0: Car 0 (^ ^) exiting
Date and Time: Mon Nov 13 08:44:34 2023
Time 3.2: Car 2 (^ <) arriving
Date and Time: Mon Nov 13 08:44:34 2023
Time 3.2: Car 2 (^ <) crossing
Date and Time: Mon Nov 13 08:44:34 2023
Time 3.4: Car 3 (v v) arriving
Date and Time: Mon Nov 13 08:44:35 2023
Time 3.9: Car 1 (^ ^) exiting
Date and Time: Mon Nov 13 08:44:35 2023
Time 4.1: Car 4 (v >) arriving
Date and Time: Mon Nov 13 08:44:35 2023
Time 4.3: Car 5 (^ ^) arriving
Date and Time: Mon Nov 13 08:44:36 2023
Time 5.6: Car 6 (> ^) arriving
Date and Time: Mon Nov 13 08:44:37 2023
Time 5.8: Car 7 (< ^) arriving
Date and Time: Mon Nov 13 08:44:37 2023
Time 6.2: Car 2 (^ <) exiting
Date and Time: Mon Nov 13 08:44:37 2023
Time 6.2: Car 3 (v v) crossing
Date and Time: Mon Nov 13 08:44:37 2023
Time 6.2: Car 4 (v >) crossing
Date and Time: Mon Nov 13 08:44:39 2023
Time 8.2: Car 3 (v v) exiting
Date and Time: Mon Nov 13 08:44:40 2023
Time 9.2: Car 4 (v >) exiting
Date and Time: Mon Nov 13 08:44:40 2023
Time 9.2: Car 5 (^ ^) crossing
Date and Time: Mon Nov 13 08:44:42 2023
Time 11.2: Car 5 (^ ^) exiting
Date and Time: Mon Nov 13 08:44:42 2023
Time 11.2: Car 6 (> ^) crossing
Date and Time: Mon Nov 13 08:44:45 2023
Time 14.2: Car 6 (> ^) exiting
Date and Time: Mon Nov 13 08:44:45 2023
Time 14.2: Car 7 (< ^) crossing
Date and Time: Mon Nov 13 08:44:46 2023
Time 15.2: Car 7 (< ^) exiting
```

Threads:

Highlighted Code:

```
pthread_t threads[totalCarNumb];
pthread_attr_t attr;
// ...
for (int i = 0; i < totalCarNumb; i++)
{
pthread_create(&threads[i], &attr, startCrossing, (void *)&newCar[i]);
}
// ...
for (int i = 0; i < totalCarNumb; i++)
{
pthread_join(threads[i], NULL);
}
```

- The code creates threads using pthread_create in a loop, and each thread executes the startCrossing function.
- The pthread_join function is used to wait for each thread to finish before the program exits.

Head-of-Line Blocking:

Highlighted Code:

```
pthread_mutex_t updateLock[4];
pthread_mutex_t turnLock[4];
sem_t enterSem[4];
sem_t exitSem;
```

Mutexes (updateLock, turnLock) and semaphores (enterSem, exitSem) are used to control access to the intersection and manage the order in which cars arrive, cross, and exit.

Turning Left :

Highlighted Code:

```
pthread_mutex_t updateLock[4];
pthread_mutex_t turnLock[4];
sem_t enterSem[4];
sem_t exitSem;
```

- If the source and destination are the same, the car is going straight through the intersection.
- The code acquires the necessary locks and semaphores to ensure proper flow through the intersection.

Driving Through :

Highlighted Code:

```
if (source == destination)
{
// ...
}
```

- If the source and destination are the same, the car is going straight through the intersection.
- The code acquires the necessary locks and semaphores to ensure proper flow through the intersection.

Turning Right :

Highlighted Code:

```
else if (isRightTurn(source, destination) == true
)
```

```
{
sem_wait(&enterSem[destination]);
crossing[destination]++;
}
```

- The isRightTurn function is used to determine if a car is turning right.
- When turning right, the code waits for the appropriate semaphore to allow entry and increments the crossing count.