**Question 1 :**

**Le bloc 1** de la fonction Main est utilisé pour empêcher l'exécution simultanée de plusieurs instances du programme en utilisant un mécanisme de mutex. Voici une explication détaillée de son rôle :

1. Mutex Initialization (Mutex) : Un objet de type Mutex est créé pour contrôler l'accès concurrentiel au bloc de code qui suit. Le premier paramètre de la classe Mutex est true, ce qui signifie que le thread actuel qui crée le mutex a le droit d'y accéder immédiatement.

2 - Mutex Check (\_prevInstance) : La variable \_prevInstance est un booléen qui indique si une instance antérieure du programme est déjà en cours d'exécution. Si \_prevInstance est false, cela signifie qu'une autre instance est déjà en cours d'exécution, et dans ce cas, le programme retourne, évitant ainsi l'exécution simultanée de plusieurs instances.

3 - Return Statement : Si \_prevInstance est false, le programme atteint la déclaration return, ce qui signifie que le programme se termine immédiatement sans exécuter le reste du code dans la fonction Main.

En ce qui concerne les améliorations, voici quelques suggestions :

Utilisation de using pour le Mutex : Vous devriez utiliser la déclaration using pour garantir que le Mutex est correctement libéré, même en cas d'exception. Cela peut être fait de la manière suivante :

|  |
| --- |
| using (Mutex mutex = new Mutex(true, AppDomain.CurrentDomain.FriendlyName, out \_prevInstance))  {  if (!\_prevInstance)  {  return;  }  // Reste du code ici  } |

**Le bloc 2** de la fonction Main est responsable de la récupération du nombre de voitures à créer à partir des arguments de la ligne de commande (args). Voici une explication détaillée de son rôle :

Parsing d'argument (int.TryParse) : Le code tente de convertir le premier argument de la ligne de commande en un entier (int). Si cette conversion réussit, la valeur convertie est stockée dans la variable \_nb\_cars.

Fallback Value : Si la conversion échoue (par exemple, si aucun argument n'est fourni ou si l'argument n'est pas un entier valide), le code attribue une valeur par défaut de 100 à la variable \_nb\_cars.

Gestion d'Exceptions : Une clause catch générique est utilisée pour capturer toutes les exceptions qui pourraient survenir pendant le processus de conversion ou d'accès à l'argument. Si une exception se produit, la variable \_nb\_cars est également initialisée à 100.

En ce qui concerne les améliorations, voici quelques suggestions :

Validation des Arguments : Avant d'essayer de convertir l'argument en un entier, il peut être utile de vérifier si au moins un argument est fourni. Cela évitera une IndexOutOfRangeException.

|  |
| --- |
| if (args.Length > 0)  {  if (!int.TryParse(args[0], out \_nb\_cars))  {  Console.WriteLine("Invalid argument. Using default value.");  \_nb\_cars = 100;  }  }  else  {  Console.WriteLine("No argument provided. Using default value.");  \_nb\_cars = 100;  } |

Utilisation de int.Parse avec gestion d'exception : Si vous souhaitez qu'une exception soit levée en cas d'échec de la conversion, vous pourriez utiliser int.Parse plutôt que int.TryParse et gérer l'exception.

|  |
| --- |
| try  {  \_nb\_cars = int.Parse(args[0]);  }  catch (FormatException)  {  Console.WriteLine("Invalid argument format. Using default value.");  \_nb\_cars = 100;  }  catch (IndexOutOfRangeException)  {  Console.WriteLine("No argument provided. Using default value.");  \_nb\_cars = 100;  } |

**Le bloc 3** de la fonction Main utilise un objet ManualResetEvent pour attendre que tous les threads de voiture (carThreads) aient terminé leur exécution avant de permettre au programme principal de se terminer. Voici une explication détaillée de son rôle :

Boucle d'attente (doneEvent.WaitOne()) : Le programme utilise la méthode WaitOne du doneEvent pour bloquer le thread principal jusqu'à ce que tous les threads de voiture aient signalé la fin de leur exécution en appelant doneEvent.Set().

En ce qui concerne les améliorations, le code semble correct pour accomplir sa tâche. Cependant, voici quelques suggestions générales :

Gestion des exceptions : Ajoutez une gestion appropriée des exceptions dans la boucle où doneEvent.WaitOne() est appelé. Si une exception se produit, assurez-vous que les ressources sont toujours libérées correctement.

Utilisation de using pour les ressources jetables : Si les classes Cl\_server\_socket et Mutex implémentent l'interface IDisposable, vous pouvez utiliser la déclaration using pour vous assurer que les ressources sont correctement libérées, même en cas d'exception.

Question 2 :

1. Oui, dans tous les cas, le code utilise le protocole IP grâce à AddressFamily.InterNetwork.
2. Non Le code actuel de la classe Cl\_server\_socket met en place un serveur socket qui écoute sur une adresse IP spécifique (127.0.0.1, soit localhost) et un port (8888). Ceci est adapté pour les connexions sur la même machine locale (loopback).

Si vous souhaitez permettre à des consoles déportées sur des ordinateurs du même réseau d'accéder à ce serveur socket, vous devrez effectuer quelques modifications :

Utilisez une adresse IP accessible depuis le réseau local : Remplacez l'adresse IP "127.0.0.1" par l'adresse IP de votre machine sur le réseau local.

1. Non Le code actuel du serveur socket est configuré pour gérer une seule connexion cliente à la fois. Il utilise serverSocket.Accept() pour accepter une connexion client, puis traite les commandes de cette connexion dans la boucle while (true).

Si vous souhaitez prendre en charge simultanément 10 consoles distantes, vous devrez ajuster votre code pour gérer plusieurs connexions en parallèle. Vous pouvez le faire en utilisant des threads ou des tâches pour chaque connexion cliente.

Quzstion3 :

Le code que vous avez fourni pour gérer la barrière dans la classe Cl\_parking semble bien structuré et utilise correctement les verrous (lock) pour assurer la synchronisation entre les threads accédant aux ressources partagées. Cependant, j'ai remarqué une petite amélioration que vous pourriez envisager.

Dans la méthode Entrance\_gate, vous avez un problème potentiel lors de l'attente d'un parking vacant. Le code actuel utilise une boucle while avec Thread.Sleep(1) pour attendre qu'une place de parking soit disponible. Cependant, cela peut être intensif en termes de ressources et n'est pas la meilleure approche.

Voici une proposition pour améliorer cette partie du code en utilisant un Monitor.Wait et Monitor.Pulse pour une attente plus efficace et moins gourmande en ressources :

Avec cette modification, la méthode Entrance\_gate utilise Monitor.Wait pour libérer le verrou et attendre qu'une notification soit reçue lorsque la condition est satisfaite. Cela réduit le besoin de boucle while avec Thread.Sleep et peut améliorer les performances et la gestion des ressources.

2 -

Les quatre variables membres de la classe Cl\_parking que vous avez mentionnées (bLock\_Save\_state\_parking\_spot, bLock\_vacant\_parking\_spot, bLock\_entrance\_gate, bLock\_exit\_gate) sont utilisées pour créer des verrous (lock) afin de synchroniser l'accès concurrent aux différentes sections de code de la classe.

L'utilisation de ces verrous est justifiée lorsque plusieurs threads peuvent accéder simultanément à des parties du code qui utilisent des données partagées, comme c'est le cas ici. Chaque verrou est associé à un contexte spécifique, comme la gestion des places de parking, la sortie de la barrière, etc.

Il est nécessaire d'avoir ces verrous distincts pour assurer une synchronisation correcte et éviter les problèmes de concurrence. Si un seul verrou était utilisé pour toutes les sections critiques, cela pourrait entraîner des problèmes de performances et de contention, car chaque opération sur les places de parking, l'entrée ou la sortie de la barrière serait bloquée pour les autres opérations.

En résumé, la création de ces quatre variables pour les verrous distincts est justifiée et contribue à garantir une gestion concurrente sûre et efficace de la classe Cl\_parking.

Question 4 :

La classe Cl\_stat\_CSV est conçue comme un singleton pour garantir qu'une seule instance de la classe est créée, et elle a un mécanisme pour créer un fichier journalier unique. Cela est réalisé en utilisant un verrou (lock) pour éviter les problèmes de concurrence lors de la création de l'instance.

Améliorations :

1. Utilisation de Path.Combine pour la construction du chemin du fichier : Utilisez Path.Combine pour construire le chemin du fichier au lieu de la concaténation manuelle de chaînes. Cela garantit la création d'un chemin correct, indépendamment du système d'exploitation.
2. Utilisation de try...finally pour libérer le verrou : Utilisez try...finally pour vous assurer que le verrou est libéré même en cas d'exception. Cela garantit que le verrou est toujours libéré, même si une exception se produit pendant l'écriture dans le fichier.
3. Gestion des erreurs pendant la création du fichier : Ajoutez une gestion des erreurs pendant la création du chemin du fichier, par exemple, si le dossier n'existe pas.