

Threading Synchronization v1.0

Introduction

You will build a car park simulation where many cars arrive concurrently. The car park has a fixed capacity. Cars must wait outside the gate in FIFO order when the park is full and only enter when a spot becomes available.

This exercise focuses on .NET framework primitives for async/threading:

- Task / async / await
- Channel<T> for a thread-safe FIFO queue
- SemaphoreSlim to model available spots
- Interlocked for atomic counters

Prerequisites

To complete this exercise, you must:

- Done the MergeSort exercise

Goal

- Create a multi-threaded/asynchronous application (many concurrent tasks)
- Use framework classes for synchronization (Channel, SemaphoreSlim)
- Explain thread safety and demonstrate atomic state with Interlocked
- Reason about potential issues like data races (optional extensions)

Provided behavior (what the finished program should do)

1. Cars arrive at random times.
2. Cars request entry via `CarPark.EnterAsync(carId, ct)`
3. If the park is full, cars wait in FIFO order.
4. Manager admits cars one-by-one as spots become free.
5. Cars park for a while and leave using `CarPark.Exit(carId, spotId)`
6. Statistics show entered/exited/inside counts and inside must never exceed capacity.

Exercise 1 IFO Arrival Queue (ChannelArrivalQueue)

File: **ChannelArrivalQueue.cs**

Implement:

Threading Synchronization v1.0

- `EnqueueAsync(...)` → write request to channel
- `ReadAllAsync(...)` → return `ReadAllAsync(ct)`
- `Complete()` → `TryComplete()`

Expected outcome: Manager sees cars in the same order they call `EnterAsync`.

Exercise 2 Capacity Gate (`SemaphoreCapacityGate`)

File: **`SemaphoreCapacityGate.cs`**

Implement:

- `WaitAsync(ct)` → wait for a spot asynchronously (no blocking)
- `Release()` → release a spot

Expected outcome: The number of cars inside never exceeds capacity.

Exercise 3 Spot Pool (`ConcurrentSpotPool`)

File: **`ConcurrentSpotPool.cs`**

Implement:

- `TakeSpot()` → dequeue a spot id; if none, throw `InvalidOperationException` (logic error)
- `ReturnSpot(spotId)` → enqueue spot id back

Expected outcome: Cars receive spot IDs 1..capacity and those IDs recycle correctly.

Exercise 4 - Atomic Stats (`AtomicStats`)

File: **`AtomicStats.cs`**

Implement:

- `MarkEntered()` → `Interlocked.Increment(ref _entered)`
- `MarkExited()` → `Interlocked.Increment(ref _exited)`

Expected outcome: Stats are stable under concurrency.

Exercise 5 - The Car Park Coordinator (`CarPark`)

File: **`CarPark.cs`** (this is the “main” threading logic)

Implement:

- `EnterAsync(carId, ct)`

Threading Synchronization v1.0

- Create a `TaskCompletionSource<int>(RunContinuationsAsynchronously)`
 - Enqueue an `EntryRequest(carId, tcs)`
 - `Await tcs.Task.WaitAsync(ct)` and return spot
- `Exit(carId, spotId)`
 - Return spot to pool
 - Release capacity gate
 - Update exit stats
 - Log something useful
- `RunManagerAsync(ct)`
 - `await foreach` over `_arrivals.ReadAllAsync(ct)`
 - For each request:
 - `await _gate.WaitAsync(ct)`
 - `int spot = _spots.TakeSpot()`
 - `_stats.MarkEntered()`
 - `request.SpotTcs.TrySetResult(spot)`
 - optional delay for gate animation (`Task.Delay(100, ct)`)

Expected outcome: System runs to completion after `CompleteArrivals()` and prints consistent stats.