Async / Await

ThreadPool Starvation

CAELOPS

Le ThreadPool

- Mécanisme de gestion automatique des threads
- Contrôle le nombre de threads actifs (worker threads)
- Créer des nouveaux threads
- Réutilise les threads existants
- Recycler les threads inactifs

CAELOPS

Worker Threads

- Notre code s’exécute au sein d’un Thread

CAELOPS

Appel synchrone

- 1 Thread exécute cet appel
- Le Thread est bloqué jusqu'à la fin de l'appel
  - Il ne peut pas être réutilisé

```
[HttpGet("sync")]
0 références
public ActionResult DelaySync()
{
    _sqlDelayService.Execute();
    return Ok();
}
```

CAELOPS

Appel asynchrone avec Task.Run

- 1 Thread exécute cet appel
  - Il est libéré lors du await
- 1 Thread exécute l'action dans le Task.Run()
  - Il est bloqué jusqu'à la fin de l'appel
- 1 Thread exécute la suite

```
[HttpGet("taskrun")]
0 références
public async Task<ActionResult> DelayTaskRunOnSync()
{
    await Task.Run(() => _sqlDelayService.Execute());
    LogThreadPoolStats();
    return Ok();
}
```

CAELOPS

Sync over async

- Appeler une méthode asynchrone de manière synchrone
  - ExecuteAsync().Wait()
  - ExecuteAsync().Result
  - ExecuteAsync().GetAwaiter().GetResult()
  - Task.Run(() => ExecuteAsync().GetAwaiter().GetResult()) .GetAwaiter().GetResult()
  - …

- Le Thread est bloqué jusqu'à la fin de l'appel
  - Génère des deadlocks dans le cas

  de synchronisation de contexte (WPF, WinForm, …)

```
[HttpGet("wait")]
0 références
public ActionResult DelayAsyncWait()
{
    _sqlDelayService.ExecuteAsync().Wait();
    return Ok();
}
```

CAELOPS

Les E/S systèmes

- Accès à la base de données
- Accès au système de fichiers
- Requêtes http,
- …

CAELOPS

ThreadPool et E/S asynchrone

- I/O Completion Port Threads
  - Utilisés pour notifier la fin d'opérations d'E/S asynchrones

CAELOPS

Appel d'E/S asynchrone

- 1 Worker Thread exécute cet appel
  - Il est libéré lors du await

```
[HttpGet("async")]
0 références
public async Task<ActionResult> DelayAsync()
{
    await _sqlDelayService.ExecuteAsync();
    return Ok();
}
```

```
public async Task ExecuteAsync()
{
    using SqlConnection connection = new(connectionString);
    using SqlCommand command = connection.CreateCommand();
    command.CommandText = procStockName;
    command.CommandType = CommandType.StoredProcedure;
    await connection.OpenAsync();
    await command.ExecuteNonQueryAsync();
}
```

- 1 Worker Thread exécute le début de l'appel et initie l'opération asynchrone d'E/S « OpenAsync »
  - Il est libéré et le threadpool utilise les threads d'E/S de completion port
- 1 Worker Thread initie l'opération asynchrone d'E/S « ExecuteNonQueryAsync »
  - Il est libéré et le threadpool utilise les threads d'E/S de completion port
- 1 Worker Thread exécute la suite du code

CAELOPS

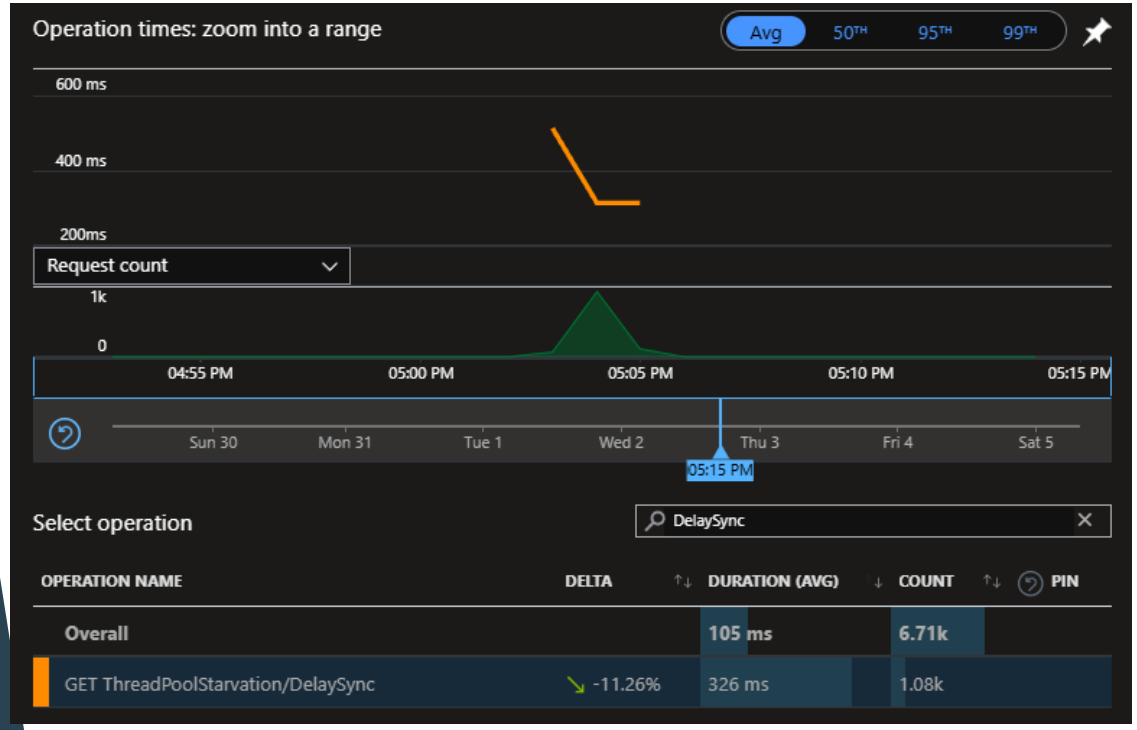
Symptômes du ThreadPool Starvation (1)

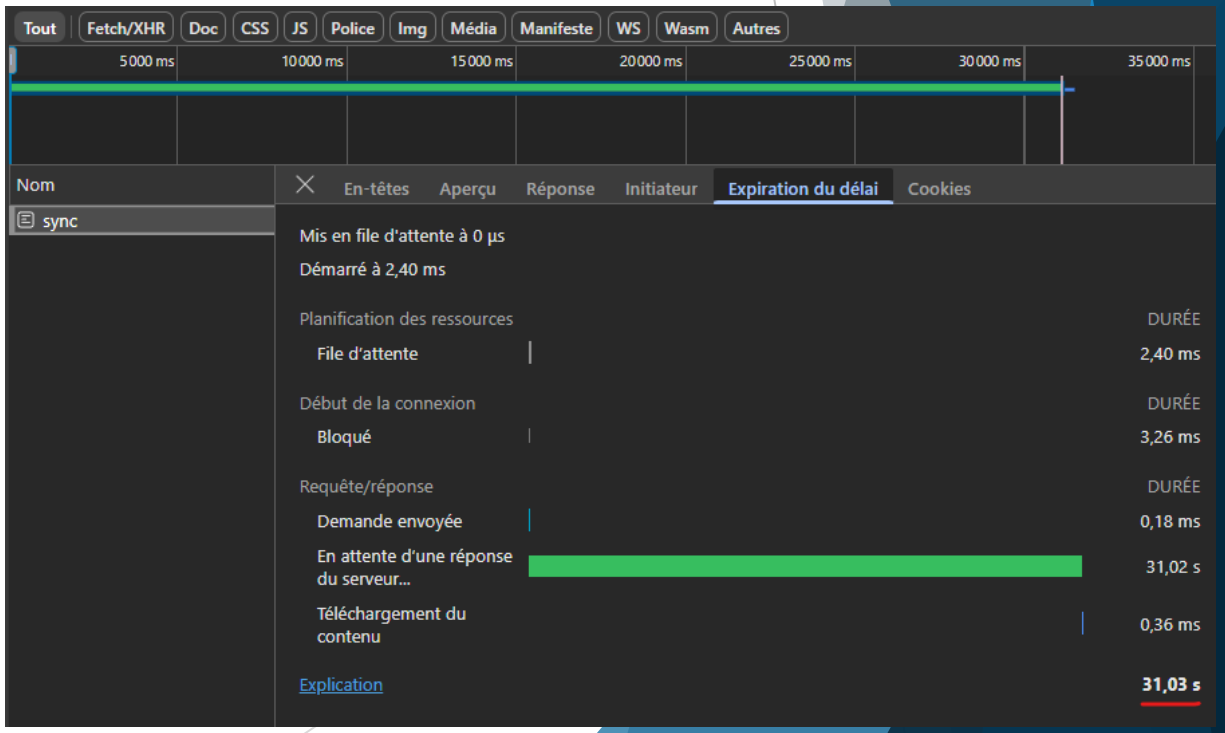
- Temps de réponse dégradés pour les utilisateurs
- Pas de consommation mémoire anormale
- Pas de consommation CPU anormale
- Temps de réponse aux API plus ou moins correctes d’après la télémétrie
- File d’attente de Work Items dans le ThreadPool augmente

CAELOPS

Symptômes : La télémétrie semble correcte alors que les utilisateurs se plaignent

- La télémétrie indique un temps de réponse moyen de 320ms alors que certains utilisateurs attendent pendant 30 secondes la réponse à l'appel

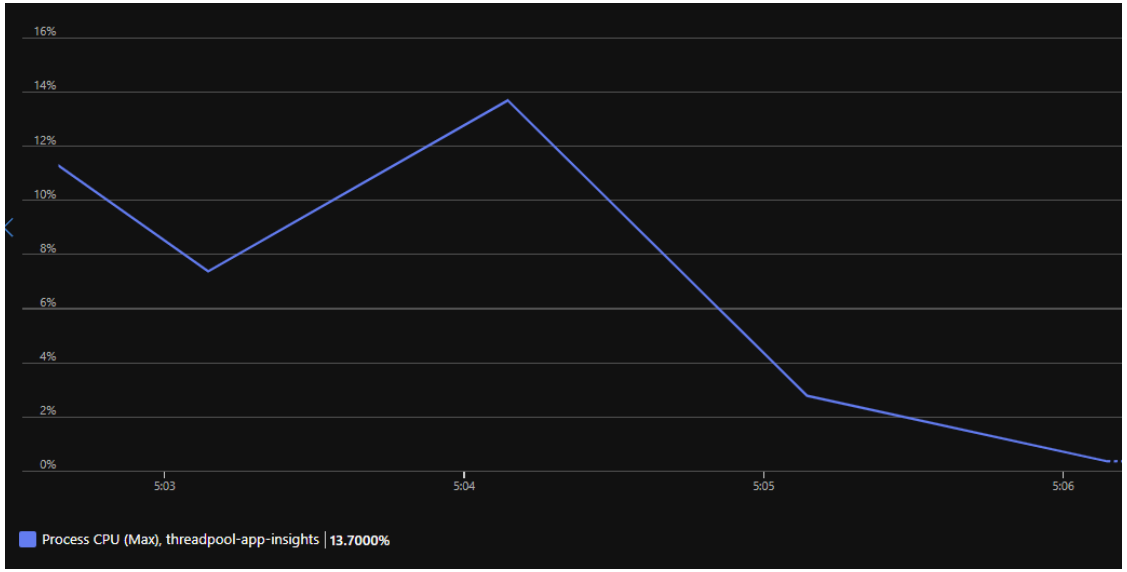


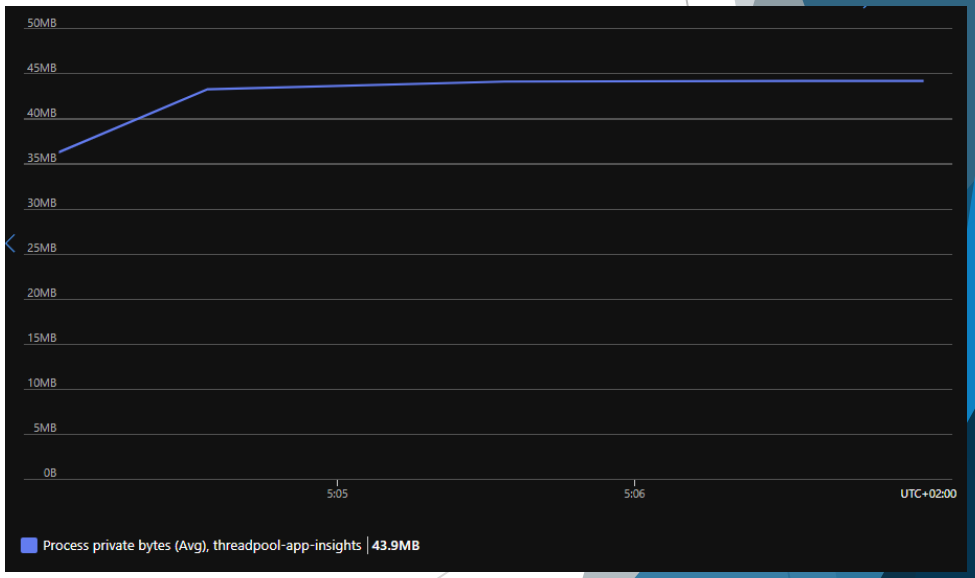


CAELOPS

Symptômes : Les ressources serveur sont correctes

- La consommation CPU Max est à moins de 15% et aucune consommation mémoire anormale n’est observée

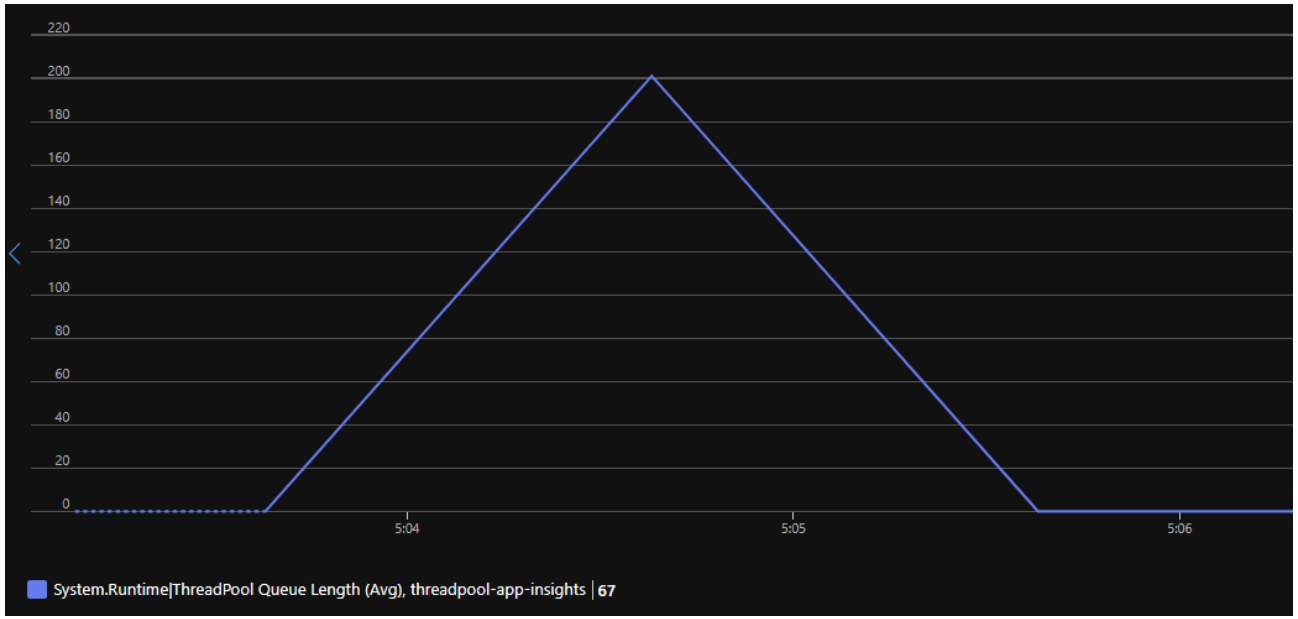




CAELOPS

Symptômes : Le ThreadPool sature

- La saturation du ThreadPool s'observe via sa file d'attente de Work Item



CAELOPS

Monitorer la file d'attente de Work Items

- Avec dotnet-counters

```
dotnet.gc.last_collection.memory.committed_size (By)          37 363 712
dotnet.gc.pause.time (s)                                            0,04
dotnet.jit.compilation.time (s)                                     2,39
dotnet.jit.compiled_il.size (By)                                 831 552
dotnet.jit.compiled_methods ({method})                             9 798
dotnet.monitor.lock_contentions ({contention})                       320
dotnet.process.cpu.count ({cpu})                                      16
dotnet.process.cpu.time (s)
    cpu.mode
    system                                                          25,79
    user                                                             1,23
dotnet.process.memory.working_set (By)                          1,5085e+0
dotnet.thread_pool.queue.length ({work_item})                    1 830
dotnet.thread_pool.thread.count ({thread})                          68
dotnet.thread_pool.work_item.count ({work_item})                 2 048
dotnet.timer.count ({timer})                                         9
```

- Avec les compteurs de performance dans Azure Application Insights

```
builder.Services.ConfigureTelemetryModule<EventCounterCollectionModule>((module, o) =>
{
    module.Counters.Add(
        new EventCounterCollectionRequest("System.Runtime", "threadpool-completed-items-count"));
    module.Counters.Add(
        new EventCounterCollectionRequest("System.Runtime", "threadpool-queue-length"));
    module.Counters.Add(
        new EventCounterCollectionRequest("System.Runtime", "threadpool-thread-count"));
});
```

CAELOPS

Documentations

- Async/Await - Best Practices in Asynchronous Programming | Microsoft Learn
- ASP.NET Core Best Practices | Microsoft Learn
  - Avoid blocking calls
  - Optimize data access and I/O
  - Return IEnumerable<T> or IAsyncEnumerable<T>
  - Complete long-running Tasks outside of HTTP requests
  - Avoid synchronous read or write on HttpRequest/HttpResponse body
  - Prefer ReadFormAsync over Request.Form
- Debug ThreadPool Starvation - .NET | Microsoft Learn
- Efficient Querying - EF Core | Microsoft Learn
  - Asynchronous programming

CAELOPS

Documentations (2)

- AspNetCoreDiagnosticScenarios/AsyncGuidance.md at master · davidfowl/AspNetCoreDiagnosticScenarios

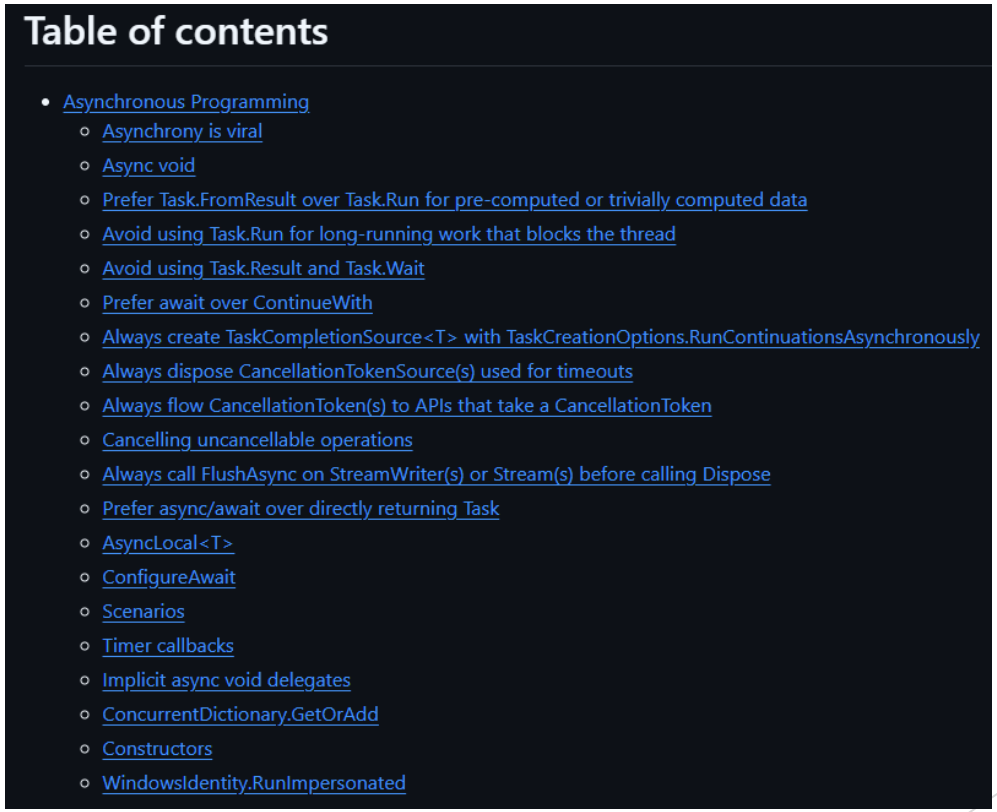


Table of contents

- Asynchronous Programming
  - Asynchrony is viral
  - Async void
  - Prefer Task.FromResult over Task.Run for pre-computed or trivially computed data
  - Avoid using Task.Run for long-running work that blocks the thread
  - Avoid using Task.Result and Task.Wait
  - Prefer await over ContinueWith
  - Always create TaskCompletionSource<T> with TaskCreationOptions.RunContinuationsAsynchronously
  - Always dispose CancellationTokenSource(s) used for timeouts
  - Always flow CancellationToken(s) to APIs that take a CancellationToken
  - Cancelling uncancellable operations
  - Always call FlushAsync on StreamWriter(s) or Stream(s) before calling Dispose
  - Prefer async/await over directly returning Task
  - AsyncLocal<T>
  - ConfigureAwait
  - Scenarios
  - Timer callbacks
  - Implicit async void delegates
  - ConcurrentDictionary.GetOrAdd
  - Constructors
  - WindowsIdentity.RunImpersonated