



Project

G.A.M.E.D.E.V.

Concurrent IO

Documentation

Contents

1	Introduction	2
2	Documentation	3
2.1	Setup	3
2.2	Supported methods	3
2.3	Tests	5
3	Advanced	6
3.1	Locking mechanism	6
3.2	IO	6
4	Support	7

Section 1

Introduction

The Concurrent IO package is a lightweight wrapper library that makes the default `System.IO.File` and `System.IO.Directory` namespaces thread-safe. By using this package, you can modify many files and directories simultaneously (concurrently, i.e. from many threads at the same time) without ever worrying about thread safety.

The following document serves as an offline documentation, briefly describing the necessary setup of the package, as well as how to use it in your Unity projects.

Section 2

Documentation

The following section describes how to import, set up, and use the Concurrent IO package in your Unity project.

For detailed information about the API, consult the API document included in the package, visit the [Project G.A.M.E.D.E.V. documentation website](#), or read the comments in the source files.

2.1 Setup

Firstly, import the package in your project like any other package on the Unity Asset Store – go to **Window\Package Manager**, select **My Assets** as the package source, and import the Concurrent IO package.

In case you are using assembly definition files for your scripts, add the Concurrent IO assembly located at **Project GAMEDEV\Tools\IO\Scripts** as an **assembly definition reference** to your assembly files. Do this for all assemblies containing scripts that need to use the `ProjectGamedev.IO` namespace.

That's it! Since the Concurrent IO package is a code library, there are no in-scene components. Simply replace your `System.IO.File` and `System.IO.Directory` calls with the `ProjectGamedev.IO` namespace. Your IO code is now thread-safe and optimized to work with many files and directories simultaneously!

2.2 Supported methods

Bear in mind not all methods from the `System.IO.File` and `System.IO.Directory` are supported. You can open **File.cs** and **Directory.cs** located in **Project GAMEDEV\Tools\IO\Scripts** to see which methods are supported, as well as their extensive documentation.

Supported methods from `System.IO.File`:

- `AppendAllLines`, `AppendAllText`, `Copy`, `Create`, `Delete`, `Exists`, `GetAttributes`, `GetCreationTime`, `GetCreationTimeUtc`, `GetLastAccessTime`, `GetLastAccessTimeUtc`, `GetLastWriteTime`, `GetLastWriteTimeUtc`, `Move`, `ReadAllBytes`, `ReadAllLines`, `ReadAllText`, `Replace`, `SetAttributes`, `SetCreationTime`, `SetCreationTimeUtc`, `SetLastAccessTime`, `SetLastAccessTimeUtc`, `SetLastWriteTime`, `SetLastWriteTimeUtc`, `WriteAllBytes`, `WriteAllLines`, `WriteAllText`.

Unsupported methods from `System.IO.File`:

- `Decrypt` and `Encrypt` – these methods are Windows-only and are no longer recommended for use; moreover, they have better alternatives in the Unity API.
- `GetUnixFileMode` and `SetUnixFileMode` – these methods are not supported on Windows.
- `OpenHandle` – this method allows non-mutex multi-threaded access to files, which is incompatible with this system.
- `Move(String, String, Boolean)` and `ResolveLinkTarget` – these methods are not available in the .NET version supported by Unity.
- `SetCreationTime(SafeFileHandle, DateTime)`, `SetCreationTimeUtc(SafeFileHandle, DateTime)`, `SetLastAccessTime(SafeFileHandle, DateTime)`, `SetLastAccessTimeUtc(SafeFileHandle, DateTime)`, `SetLastWriteTime(SafeFileHandle, DateTime)`, and `SetLastWriteTimeUtc(SafeFileHandle, DateTime)` – these overloads are not available in the .NET version supported by Unity.
- `AppendAllLinesAsync`, `AppendAllTextAsync`, `ReadAllBytesAsync`, `ReadAllLinesAsync`, `ReadAllTextAsync`, `ReadLinesAsync`, `WriteAllBytesAsync`, `WriteAllLinesAsync`, and `WriteAllTextAsync` – asynchronous methods are not compatible with the system.
- `AppendText`, `CreateSymbolicLink`, `CreateText`, `Open`, `OpenRead`, `OpenText`, `OpenWrite` – these methods return objects that allow external (not within the system) modification of the path.
- Methods that take `SafeFileHandle` instead of a string path as a parameter (e.g. `GetAttributes`) – the string path necessary for the system's internal concurrency mechanism cannot be extracted from a `SafeFileHandle` object.
- `ReadLines` – This method cannot be supported by the system, as it returns an enumerator for the file contents. This enumerator is outside of the scope of the system and, with it, the file contents can be accessed at a later time, meaning that no other thread can safely access the file.

Supported methods from `System.IO.Directory`:

- `CreateDirectory`, `Delete`, `EnumerateDirectories`, `EnumerateFiles`, `EnumerateFileSystemEntries`, `Exists`, `GetCreationTime`, `GetCreationTimeUtc`, `GetCurrentDirectory`, `GetDirectories`, `GetDirectoryRoot`, `GetFiles`, `GetFileSystemEntries`, `GetLastAccessTime`, `GetLastAccessTimeUtc`, `GetLastWriteTime`, `GetLastWriteTimeUtc`, `GetLogicalDrives`, `GetParent`, `Move`, `SetCreationTime`, `SetCreationTimeUtc`, `SetCurrentDirectory`, `SetLastAccessTime`, `SetLastAccessTimeUtc`, `SetLastWriteTime`, `SetLastWriteTimeUtc`.

Unsupported methods from `System.IO.Directory`:

- `CreateTempSubdirectory` – this method is not available in the .NET version supported by Unity.
- `CreateDirectory(String, UnixFileMode)` – `UnixFileMode` is not available in the .NET version supported by Unity.
- `CreateSymbolicLink`, `ResolveLinkTarget` – these methods return objects that allow external (not within the system) modification of the path.

2.3 Tests

The automated tests used during the development of this tool are included under **Project GAMEDEV\Tools\IO\Scripts\Tests**.

You can safely remove the entire **Tests** folder if you have no need for the tests. Alternatively, you can run them using the Unity Test Runner package, as well as expand them in case you make custom modifications to the Concurrent IO package.

Section 3

Advanced

Below are advanced details on how the Concurrent IO package works. Read this in case you need implementation specifics.

3.1 Locking mechanism

The main component of the system is the **ConcurrentIO** class. Its sole responsibility is locking delegated functions, so no 2 threads may execute them simultaneously. In order to boost performance, so-called "named" locks are used, where 2 threads may run concurrently as long as their locks have different names. This allows 2 files to be modified concurrently, but blocks concurrent modifications to a single file.

In the case of directory access, so-called "exclusive" (or mutex – mutual exclusion) locking is used, where all threads are halted, 1 thread modifies the directory, and the rest of the threads are then resumed. Since directories can contain many subdirectories and files, named locking is not feasible, as a thread could be modifying a subdirectory while another is modifying its parent directory, resulting in an access violation (even though their locks have different names).

The named locks themselves are simple empty objects stored in a concurrent dictionary collection, where the name is their key in the dictionary. The dictionary is automatically pruned when its element count reaches a certain threshold in order to prevent its size from exploding in long-running applications.

The correctness of the locking mechanisms is guaranteed by automated load testing, where thousands of threads attempt to access files and directories concurrently using the **ConcurrentIO** class.

3.2 IO

The **File** and **Directory** classes are simple wrappers that use the **ConcurrentIO** class to call various methods from the **System.IO** namespace. They do not offer additional functionality beyond that.

Section 4

Support

In case you experience any problems with the tool, cannot manage to set it up properly, or want to simply propose an improvement, do not hesitate to contact us at contact@project-gamedev.com, or fill out the form at project-gamedev.com/contact.

Best of luck with your game development journey!