GZipArchiver

Author: Artur Nasyrov

Introduction

The solution consists of 3 parts:

- the library project GZipArchiver.Lib
- the unit test project GZipArchiver.Lib.Tests
- the console app GZipArchiver.App

The first part contains the details of the console application. The next part describes the details of the library project which contains all compression and decompression logic. Then the final part contains the details of the processing algorithm.

The console application

The console application has two commands

- compressing: GZipTest.exe compress [original file name] [archive file name]
- decompressing: GZipTest.exe decompress [archive file name] [decompressing file name]

Before the command execution the application reads the application settings to create a compressor or a decompressor with the certain parameters.

The parameters are:

- workThreadCount – the number of working threads
- blockSize – the size of a file chunk to be processed
- queueInSize – the capacity of the input queue
- queueOutSize – the capacity of the output array

The library project

Inside the library there are several large packages and a few small ones. Also it includes the compressor and the decompressor classes which are out of any package.

GZipArchiver.Lib packages

- **Collections** – the package contains a concurrent queue `SyncQueue` and a concurrent array `SyncTapeArray`. These collections are used by operations internally and are described later.
- **IO** – the package contains read and write classes**.**
- **Processors** – there are two types of processors. The `FileProcessor` is responsible for creation and orchestration of threads, operations and special work items. According to the compression mode a work item either compress or decompress a file block.
- **Factories** – a factory creates work items inside the processor.
- **Operations** – operation and context are related to the file processor.
- **Enums** – the package contains `ProcessState` – the result enumerator.
- **Threads** – each thread is designed to execute its own operation and save its result.

The next part contains some details about classes from the mentioned packages.

File compressor and decompressor

The compressor or decompressor process an input file and provide public method `Process` to be called in the application. After the execution, the application will receive `OperationResult` which contains a result of the operation and some fault messages in case of error. The compressor and the decompressor use the same procession logic in most of cases which was put into `BaseFileArchiver`.

There is only difference between type of file reader, file writer and internal work processor. For example, the `RawFileReader` class is used to read an original file by chunks of the same length or the `CompressedFileReader` class is used to read compressed chunks of unknown length.

File structure

The compressed file structure Is simple. The beginning of the file is a 4 byte encoded number of compressed chunks. After this block, there is a repeating substructure of 4-byte encoded length of the next chunk and a byte array of encoded chunk content.

COUNT + array_of(CHUNK_LEN + COMPRESSED_CHUNK).

In case of an original file the number of chunks is calculated using the length of a file and the chunk size.

File processor

The file processor contains the most important processing parts. Both compressor and decompressor use it. The file processor creates one read operation, one write operations and multiple process operations. Each operation is located in own thread. Also the file processor creates the special `OperationContext` which is shared between operations.

After thread and operation were created the processor runs all threads and waits until they are finished. The file processor uses `CountdownEvent` to signal the main thread to continue. At the end, the main thread gathers all operation thread results, aggregates it and returns to the compressor or the decompressor.

Operations

As it was mentioned, there are 3 types of operation – read, process and write. Interaction between them is shown in the attachment file “process.pdf”.

Read operation

The operation reads a file using a specific reader and puts the read file chunk to the input queue. The queue has bounded capacity, so the read thread where the operation is to be performed has to wait the signal in case of the full queue. The thread receives a signal after dequeuing by one of process operations. The `SyncQueue` is implemented using `Monitor` with timeouts. Thus, if lock is not reacquired after 100 ms, the operation does not receive any file chunk and tries again, unless it has been cancelled.

Process operation

The operation uses `CompressWorkItem` or `DecompressWorkItem` internally. At the beginning the operation takes a chunk from the input queue or waits for a new chunk. After that, it converts the file chunk using an attached work item. Finally, the processed file chunk is inserted to the array which was called `SyncTapeArray`. Basically, it is makes possible to write shuffled input elements to an array in the specific order. The tape array also provides method `TryGetNextRange` of extraction of N-first elements which do not have any blank. That allows to write more elements at the same time without locking the array. Internally, the collection also uses the `Monitor`.

Write operation

The last operation. Also uses the tape array and calls `TryGetNextRange` to receive the processed chunks. The file writer then saves data to the storage. After that step, it continues taking the next range until the operation is completed or canceled.

Work item

Compression and Decompression work items are created using instance of `WorkItemFactory` which is created inside the file processor – see the method `CreateOperationsAndContext`. All work items are stored inside the file processor array and are disposed at the end of the compression or decompression process. That allows to not recreate memory streams which are used internally to read data from chunks and copy it to the Gzip stream.

The processing algorithm

This section contains the simplified version of the algorithm which was used inside the library to compress or decompress file. The details of an input file processing are depicted in the attachment file “process.pdf”.

Each step is described separately.

Input: compression mode, input file

Output: output file, result of operation

If mode = compress then

    Initialization of the file compressor

    Start the file compression

    Return a result of the file compressor execution

Else

    Initialization of the file decompressor

    Start the file decompression

    Return a result of the file decompressor execution

End if

Step “Initialization of the file compressor” or “Initialization of the file decompressor”

    Set the compression mode

    Set the file chunk size

    Set the number of process threads

    Set the capacity of the input queue

    Set the capacity of the output array

Step “Start the file compression” or “Start the file decompression”

Create the file reader

Create the file writer

Create the file processor

Start the file processor

Clear used resources during the execution of the file processor

Return a result of the file processor

Step “Start the file processor”

Create the read operation

Create the write operation

Create the array of process operations

Create the operation context

Create the read thread

Create the write thread

Create the array of process threads

Set the countdown event to the number of all threads

Start the read thread

Create the write thread

Create the array of process threads

Wait until all threads have signaled.

Return the aggregated result collected from all threads.

Step “Start the read thread”

Read the number of a file chunks.

Set the number to the operation context.

Unblock other threads.

Execute the read operation until it has finished or has been cancelled.

Return a result of the operation

Step “Start the write thread”

Block the thread until the operation context has been set.

Execute the write operation until it has finished or has been cancelled.

Return a result of the operation

Step “Start the process thread”

Block the thread until the operation context has been set.

Execute the process operation until it has finished or has been cancelled.

Return a result of the operation