**Задача:**

Консольное приложение на C# для поблочного сжатия и распаковки файлов с помощью System.IO.Compression.GzipStream

При работе с потоками использовались только базовые классы и объекты синхронизации

Параметры программы, имена исходного и результирующего файлов задаются в командной строке следующим образом:

GZipTest.exe compress/decompress [имя исходного файла] [имя результирующего файла]

В случае успеха программа должна возвращать 0, при ошибке 1.

**Реализация**

**Общее описание:**

Приложение реализует две версии алгоритма GZIP:

1. Ассинхронный вариант, в котором сжатые блоки записываются в целевой файл абсолютно независимо друг от друга, т.е. в порядке поступления в очередь записи. Данный вариант не предусматривает возможности распаковки стандартными методами – сторонними архиваторами, т.к. в процессе сжатия нарушается формат GZIP и порядок следования блоков не соответствует исходному.
2. Версия классического GZIP, реализованная поблочно, в которой порядок записи соблюдается. Результат может быть разархивирован любым сторонним архиватором.

На входе, через командную строку, программа получает наименование исходного и результирующего файлов, параметр, указывающий на требуемое действие над данными исходного файла (и в качестве необязательного параметра – указание наименования архиватора). Затем переводит эти параметры в рабочий вид и передаёт методу, реализующему чтение исходного файла, многопоточное преобразование данных и поблочную запись в результирующий файл. Во время работы, осуществляется информирование пользователя ходом процесса преобразования, а также ведётся журнал отчётности. Программа реализована легко масштабируемой и способной работать с любым архиватором, реализующим преобразование байтовых последовательностей. Для повышения эффективности и отказоустойчивости, приложение строит свою работу исходя из переменных окружающей среды.

**Технические особенности:**

Параметры командной строки передаются экземпляру класса OperatingParameters, задачей которого является осуществить чтение входных данных, опросить систему на предмет имеющихся ресурсов и конфигурации среды (свободная физическая память и разрядность системы) и сформировать группу рабочих параметров приложения. Затем данный объект передаётся созданному экземпляру класса TrafficController, отвечающему за поблочное чтение исходного файла, запись результата и запуск (а также - контроль работы) потоков, которые, в свою очередь, осуществляют непосредственное преобразование данных.

Для получения сведений о состоянии памяти была использована функция WinAPI GlobalMemoryStatusEx, возвращающая сведения об ОЗУ локальной станции в структуре MEMORYSTATUSEX (см. OperatingParameters.cs). Используемые программой параметры окружающей среды были собраны в статическом классе EnvironParameters для возможности простого доступа к ним различными объектами приложения.

Эмпирическим методом было установлено, что с точки зрения скорости работы, оптимальнее всего будет формирование количества потоков шифрования равным количеству логических процессоров локальной станции пользователя, а также выделения процесса чтения и записи в отдельные потоки.

Метод Start класса TrafficController, запускает поток чтения (рабочей функцией которого является метод ReadFStream), поток записи результата (метод WriteFStream) и потоки конвертирования данных (метод ConvertData). Также он создаёт два экземпляра класса ThreadSafeQueue (readBlocks и blocks2Write) который является потокобезопасной очередью с буферизацией.

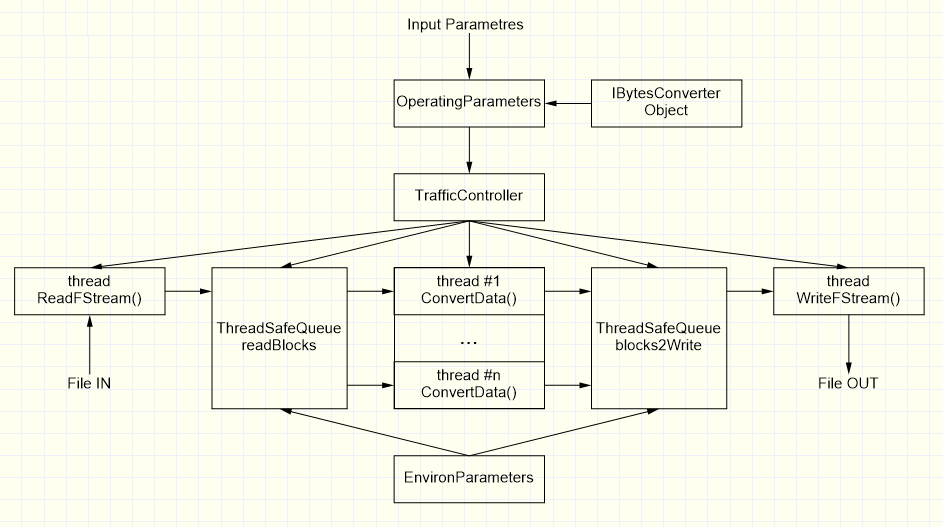
* readBlocks используется в качестве контейнера данных, получаемых от потока чтения и передаваемых в потоки конвертирования.
* blocks2Write используется в качестве контейнера данных, получаемых от потоков конвертирования и передаваемых в поток записи результата

Синхронизация потоков и потокобезопасность при работе с общими данными внутри очереди осуществляется при помощи класса Monitor пространства имён System.Threading.

* Методом Monitor.Enter осуществляется монопольная блокировка объекта
* Методом Monitor.Wait осуществляется разблокировка объекта и перевод потока в режим ожидания повторной блокировки со стороны какого либо другого потока.
* Методом Monitor.PulseAl осуществляется передача сигналов всем ожидающим потокам о смене состояния объекта.

О завершении работы потоки сигнализируют при помощи массива с объектами AutoResetEvent, (один массив соответствует потокам конвертирования, второй – операциям ввода-вывода), вызывая метод Set для элемента массива, индекс которого соответствует наименованию сигнализирующего потока (в случае с потоками конвертера). Ожидание сигналов завершения работы от потоков осуществляется методом WaitAll класса WaitHandle из пространства имён System.Threading. Работа этого метода заключется в ожидании получения сигнала всеми элементами массива с объектами класса AutoResetEvent, обозначение которого передаётся ему в качестве параметра.

Общая неформализованная схема работы программы может быть представлена следующим видом:



**Реализация. Описание работы конвертеров**:

Для возможности простого расширения приложения, механизмы архивирования представлены в виде группы классов, реализующих общий интерфейс IBytesConverter. В состав каждого класса входит 3 свойства, позволяющие идентифицировать тип класса, метод его работы и характеристику, отображающую, является ли данный алгоритм упорядоченным, т.е. требуется ли соблюдать порядок выходной информации или нет, а так же 2 метода: Метод Convert реализует процесс преобразования байт-массивов, метод GetBlockFromStream извлекает порцию данных из заданного Stream.

Принцип работы конвертера, реализующего неупорядоченное архивирование данных следующий:

Получая на входе порцию данных, конвертер производит сжатие оной, используя System.IO.Compression.GzipStream, после чего добавляет в полученный блок данных дополнительную информацию, позволяющую однозначно восстановить исходный порядок блоков в исходном файле, а также размер полученного байт-массива.

Согласно данным о GZIP из открытых источников

<http://www.onicos.com/staff/iz/formats/gzip.html>

<http://www.zlib.org/rfc-gzip.html#jl-gzip>

Имеется следующий формат:

+---+---+---+---+---+---+---+---+---+---+

|ID1|ID2|CM |FLG| MTIME |XFL|OS | (more-->)

+---+---+---+---+---+---+---+---+---+---+

........

+---+---+---+---+---+---+---+---+

| CRC32 | ISIZE |

+---+---+---+---+---+---+---+---+

Практические опыты показали, что:

1. Блок в 4 байта MTIME, в котором должно хранится время запаковки файла не несёт никакой практической нагрузки, а в случае с GzipStream не используется вовсе.
2. Блок в 4 байта ISIZE, в котором должен хранится размер исходного файла не несёт никакой функциональной нагрузки для сторонних архиваторов, поэтому первоначально была реализована запись порядкового номера блока в исходной последовательности данных в эту область для того, чтобы не нарушать формат архива.

Однако в случае с реализацией GZIP функционалом .Net из пространства System.IO.Compression, если в ISIZE не хранится размер исходного блока данных, то для распаковки такого блока требуется указать размер исходных данных, т.к. попытка скопировать данные из GZipStream куда-то без указания явного размера выдаст ошибку о несоответствии размера последовательности заявленному.

Поэтому для возможности поблочного разархивирования:

1. В блок MTIME вносятся данные о размере сжатого блока (т.к. поле 4 байта, то ограничение на размер блока получается в 4294967295 байт (или 4.29гб по системе МЭК 1998г), что больше ограничения в 2гб памяти на размер объектов в CLR вне зависимости от разрядности исполняемой среды
2. Порядковый номер исходного блока в исходном файле (используя 4 байта) вносится в начало последовательности таким образом:   
   Размер массива увеличивается на 2 байта вначале, копируется туда 4 байта индекса блока, затерев тем самым ID1, ID2 т.к. в GZIP они имеют константные значения

Данные, поступающие от конвертеров в очередь записи записываются в выходной файл в порядке их поступления от потоков, исходный порядок не соблюдается.

Процесс разархивирования, реализованный в классе GzipDecompressorAsync, производит обратные действия – при считывании блока из потока исходного файла, сначала он считывает 4 байта содержащие порядковый номер исходного блока данных в исходном файле, затем восстанавливает префикс GZIP (2 байта секций ID1 и ID2), затем из секции MTIME прочитывает размер архивированного блока в потоке Stream, инициализирует массив байтов и помещает в него считанные данные. Т.к. для процесса восстановления данных требуется соблюдать их порядок в исходном файле, то очередь имеет одну особенность в реализации.

**Особенности работы очереди ThreadSafeQueue**

Т.к. согласно условиям задачи – в реализации GzipCompressorAsync блоки обрабатываться и записываться на диск независимо друг от друга, то при обратном конвертировании необходимо восстановить порядок блоков, как в исходном файле. Поэтому в случае обратной работы, очередь на запись заполняется потоками в порядке индексов, которые соответствуют порядку блоков в исходном файле, при этом если индекс блока в потоке, который пытается добавить свой блок в очередь записи не соответствует требуемому очередью, то заставлять поток ждать своей очереди нельзя, т.к. возможна ситуация, при которой все текущие потоки не будут содержать требуемого индекса и все зависнут в ожидании. Поэтому в ThreadSafeQueue реализован так называемый буфер, в который потоки “складывают” блоки, индекс которых не соответствует ожидаемому очередью для временного хранения. После успешного же добавления элемента в очередь, очередь проверяет буфер на соответствие элементов условиям требуемого следующего индекса и в случае удачи – изымает элемент из буфера и добавляет в себя.

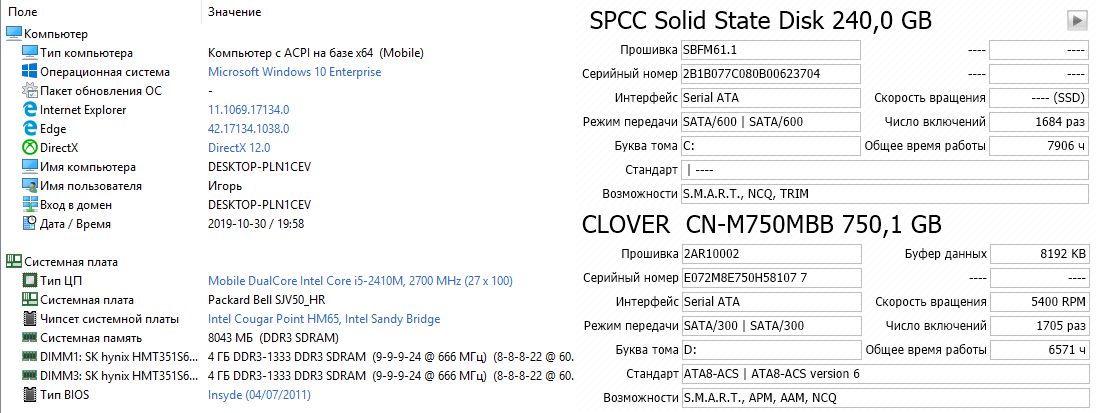
Для реализации очереди используется класс Queue<T> Class из System.Collections.Generic, который реализован разработчиками в виде одномерного массива фиксированной длины, при этом при добавлении элемента (Enqueue(T item)), размерность массива переопределяется при необходимости, а при удалении – не изменяется без вызова специального метода.

А для реализации буфера используется класс SortedList<TKey,TValue> пространства имён System.Collections.Generic, который реализован в виде двух массивов фиксированной длины TKey[] и TValue[] values, размерности которых так же, как и в случае с очередью – изменяются при необходимости при добавлении новых элементов и не изменяются при удалении, при этому сортировка происходит на стадии добавления нового элемента при помощи Array.BinarySearch<TKey>.

Иными словами использование SortedList более энергозатратно по параметру скорости работы с содержимым, чем использование очереди, но в условиях текущей задачи энергозатратность данного класса нивелируется его малым практическим размером с учётом способа его использования.

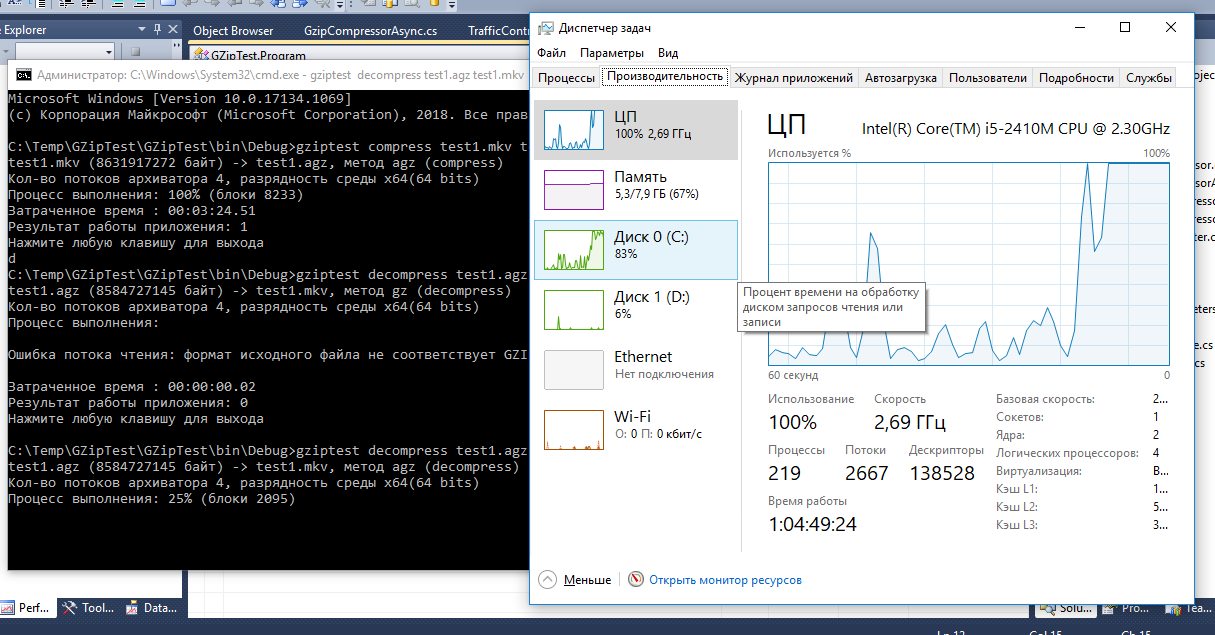
**Тестирование**

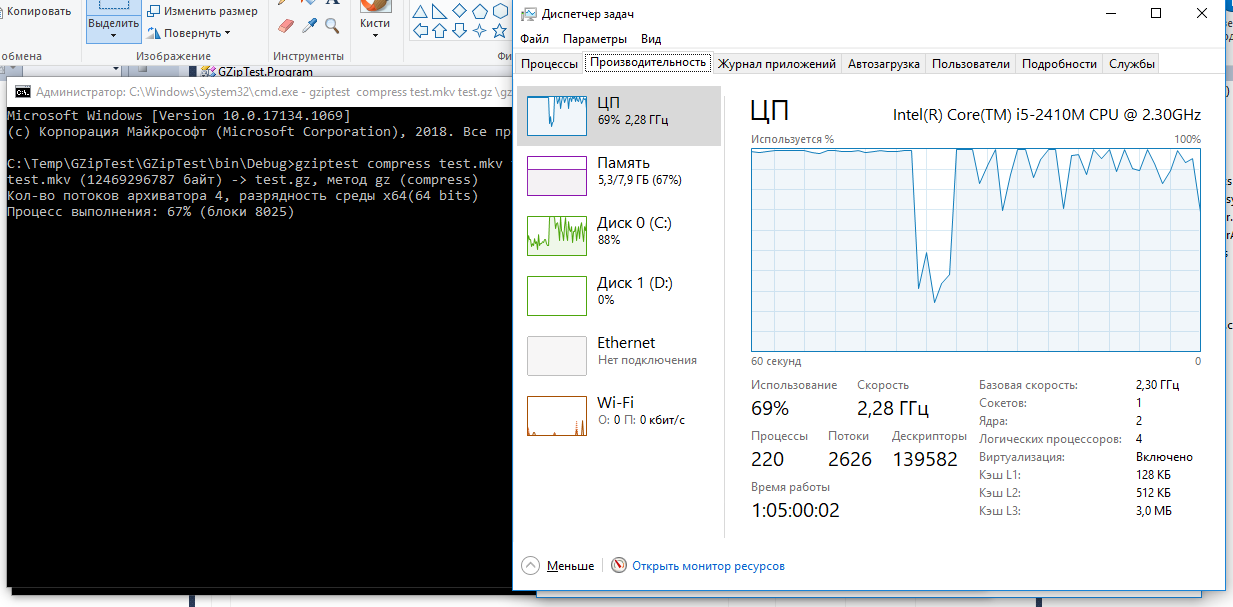
Технические данные среды:



Тестирование проводилось на файлах размером от 500 кб и до 18гб сначала на диске HDD SATA II 5400 rpm и на диске SSD SATA III.

По результатам тестирования видно, что вначале выполнения приложения нагрузка полная на используемые ресурсы CPU и жесткого диска, затем загрузка процессора идёт полностью в случае, если поток записи справляется с записью блоков и нагрузка на CPU уменьшается в случае полной загрузки работы жесткого диска.





Результаты тестов представлены в файлах .log в директории Information

Отдельно стоит отметить отчёт о разархивировании файла размером 18гб с SSD на HDD, в котором видно, что сработало ограничение на размер обеих очередей, т.к. HDD неисправен и временами “зависает”:

Дата: 31.10.2019, время: 1:07

test1.agz (18073000263 байт) -> d:\Films\test.mkv, метод agz (decompress)

Кол-во потоков архиватора 4, разрядность среды x64(64 bit)

Объём свободной физической памяти: 3964604416 байт

Ограничение на размер очереди: 2000 элементов

Количество блоков: 17374

Макс.элементов очереди: 2001, макс.элементов буфера: 0

Макс.элементов очереди: 2007, макс.элементов буфера: 18

Затраченное время : 00:12:46.22

Результат: Успешно

Это ограничение можно ещё увеличить, добавив дополнительные проверки на дисбаланс в реальном времени, т.к. задача потоков не забить свободную память, создав тем самым сильный дисбаланс в очередях, а не давать простоя потокам, разгружающим очереди – в данном случае потоку записи в результирующий файл.

