**Стовпяк Андрей**

+7 9021767966

stovpyak@gmail.com

Оглавление

[Часть 1 (от 29.08.2016) 2](#_Toc460802229)

[Алгоритм и ход рассуждений 2](#_Toc460802230)

[FilePart 2](#_Toc460802231)

[Очереди 2](#_Toc460802232)

[Обработчики очередей 3](#_Toc460802233)

[Плановое завершение работы программы 3](#_Toc460802234)

[Логирование 3](#_Toc460802235)

[Эксперименты на реальных данных 4](#_Toc460802236)

[Проверка корректности архивирования 6](#_Toc460802237)

[Разархивирование файлов 6](#_Toc460802238)

[Алгоритм разархивирования: 6](#_Toc460802239)

[Тесты 7](#_Toc460802240)

[Планы на следующие версии 7](#_Toc460802241)

[PS. 7](#_Toc460802242)

[Часть 2 (от 04.09.2016) 8](#_Toc460802243)

[Алгоритм и архитектура. 8](#_Toc460802244)

[Компрессия 8](#_Toc460802245)

[Плановое завершение работы 9](#_Toc460802246)

[Декомпрессия 9](#_Toc460802247)

[Стратегия компрессии и декомпрессии 10](#_Toc460802248)

[Ответы на замечания 10](#_Toc460802249)

[Замеры на тестовых данных 11](#_Toc460802250)

[Компрессия 11](#_Toc460802251)

[Декомпрессия 11](#_Toc460802252)

# Часть 1 (от 29.08.2016)

23.08.2016 получил тестовое задание.

30.08.2016 отправил первый вариант решения тестового задания.

## Алгоритм и ход рассуждений

Процесс архивирования состоит из 3 этапов:

1. Чтение исходного файла;
2. Архивирование;
3. Запись архива на диск.

Какие этапы можно выполнять параллельно?

Чтение исходного файла. Несколько экспериментов показали, что это возможно сделать несколькими потоками. Каждый поток открывал stream для чтения (FileAccess.Read). Устанавливал курсор на нужную позицию и читал необходимое кол-во байт.

Архивирование. Поток получает часть прочитанного файла и архивирует его. Таких потоков может быть несколько.

От распараллеливания записи на диск я отказался. Так как не придумал как одновременно несколькими потоками формировать один файл.

### FilePart

Таким образом сформировалось понятие «часть фала» FilePart.

Это сущность, которая последовательно проходит следующие этапы.

1. Инициализация;
2. Чтение;
3. Архивирование;
4. Запись.

И заново повторяем этапы с 1 по 4

Таких частей может быть несколько одновременно обрабатываемых в системе. С одной стороны, их должно быть так много, чтобы все ядра параллельно обрабатывали части, с другой стороны есть ограничение на размер допустимой памяти, которую может занимать приложение.

Фактически параметрами являются:

* Кол-во одновременно живущих в системе FilePart `ов (maxActivePartCount);
* Размер одной части.

Далее в зависимости от размеров исходного файла получаем кол-во частей, на которые будет разбит исходный файл. То есть кол-во экземпляров FilePart, созданных и «живущих» в системе ограничено, а по мере их обработки в них загружаются новые части файла.

Можно было хранить части в общем «хранилище» и менять у FilePart статусы по мере её обработки. Но от такого варианта я отказался. Вариант с несколькими очередями показался более логичным и наглядным.

### Очереди

В итоге система состоит из 4 очередей:

1. Empty. Очередь пустых FilePart. Здесь хранятся части только что созданные (в начале работы системы), а также части, которые уже прошли полный цикл обработки и готовы использоваться ещё раз.
2. ForReaders. Очередь проинициализированных FilePart. Здесь части, в которых известно с какой позиции читать файл и размер части, а также индекс части (так как записывать их нужно в таком же порядке как читали) – это части готовые для чтения.
3. ForArchivers. Очередь прочитанных FilePart. Здесь части, в которые уже загрузили часть содержимого файла – части готовые для архивирования.
4. ForWriter. Очередь заархивированных FilePart. Здесь части, в которых уже готова заархивированная часть файла – части, гтовые для записи в архив.

### Обработчики очередей

Между очередями находятся обработчики очередей (QueueHandlers).

Они извлекают FilePart из очереди (при её появлении в очереди или при своем освобождении). Выполняют необходимые действия с частью или делегируют другим частям (Workers) и помещают обработанную часть в следующую очередь.

В итоге в системе 4 обработчика очередей:

1. PartInitizlizer. Ждет части из очереди Empty. С помощью объекта Strategy инициализирует часть. В FilePart записываем положение курсора, с которого необходимо читать файл и размер части файла и индекс части. И помещает часть в очередь ForReaders.
2. ReadersRuner. Ждет части из очереди ForReaders. Создает поток для чтения - Reader. Отдает ему FilePart для чтения. И возвращается к ожиданию частей в очереди. Reader читает необходимую часть из файла, помещает её в FilePart.Source и помещает FilePart в очередь ForArchivers.
3. ArchiversRunner. Ждет части из очереди ForArchivers. Создает поток для архивирования – Archiver. Отдает ему FilePart для архивирования. И возвращается к ожиданию частей в очереди. Archiver архивирует часть файла, помещает её в Part.Result, очищает Source (так как он больше не нужен). Помещает FilePart в очередь ForWriter.
4. Writer. Ждет части из очереди ForWriter. Причем в отличие от остальных обработчиков запрашивает не любую появившуюся (точнее первую пришедшую FilePart в очереди), а FilePart со определенным индексом. Это необходимо, так как Archiver могут завершать свою работу не в том порядке в котором читали части. Далее Writer записывает FilePart.Result в файловый поток. Очищает FilePart.Result – он теперь не нужен. И помещает FilePart в очередь Empty.

### Плановое завершение работы программы

Strategy отказывается инициализировать FilePart, если все части файла уже все прочитаны. В таком случае PartInitizlizer просто выводит FilePart из работы. Когда все «живущие» FilePart выведены – это является признаком того, что файл прочитан, его части заархивированы и записаны в итоговый файл-архив. После этого PartInitizlizer сообщает основному потоку, что работа завершена. Основной поток выставляет признак окончания работы у всех QueueHandlers. И просит все очереди сообщить своим QueueHandlers что больше FilePart ждать не нужно. QueueHandlers завершают свою работу.

### Логирование

Создана абстракция ILogger с единственным методом Add(string msg)

Всем работающим частям передается ссылка на данную абстракцию.

Есть несколько реализаций данного контракта:

* LoggerDummy – ничего никуда не пишет и не запоминает. Используется в тестах
* ConsoleLogger – выводит отладочную информацию в консоль
* FileLogger – запоминает и сохраняет отладочную информацию в файл
* CompositeLogger – позволяет «сгруппировать» несколько логеров – выводить информацию сразу в несколько мест.

Таким образом можно различными способами «конфигурировать» запись отладочной информации, прозрачно для рабочих блоков.

## Эксперименты на реальных данных

IStrategy – было создано несколько реализаций стратеги:

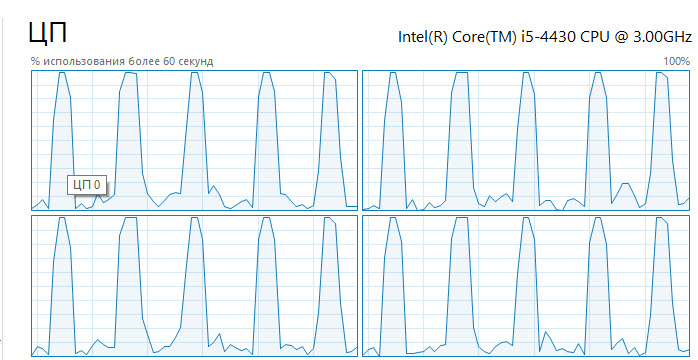
1. StrategyStub – вручную можно задать все три параметра: кол-во частей живущих в системе, размер части, количество частей файла.
2. SmartStrategy – в зависимости от параметров компьютера стратегия должна устанавливать оптимальные параметры. В качестве целевой функции было принято минимизация времени архивирования файла.

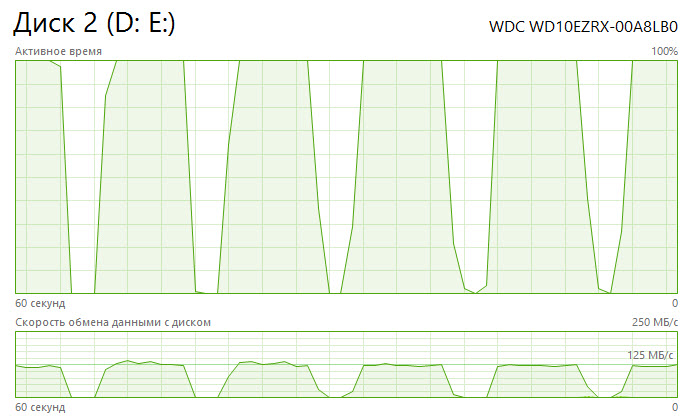
Эксперименты с использованием StrategyStub дали следующий результат:

Компьютер. 4 ядра процессора. 8 Гб оперативной памяти. Замеры на файле 10 ГБ.

Кол-во частей, одновременно живущих в системе выбралось разное.

Первые же замеры на данных показали, что графики загруженности процессора и диска чередуются и идут в противофазе. То есть когда читались данные (работали reader) процессор простаивал (arhiver ждали частей), а когда части начинали архивироваться загрузка 4 ядер 100% - простаивали reader`ы.





Также время работы reader`ов было (maxActivePartCount = 5, partCount = 50)

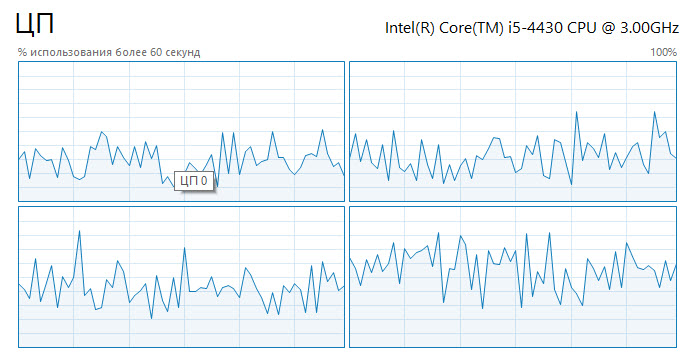
* 2209 ms
* 3929 ms
* 5587 ms
* 8108 ms
* 9058 sm

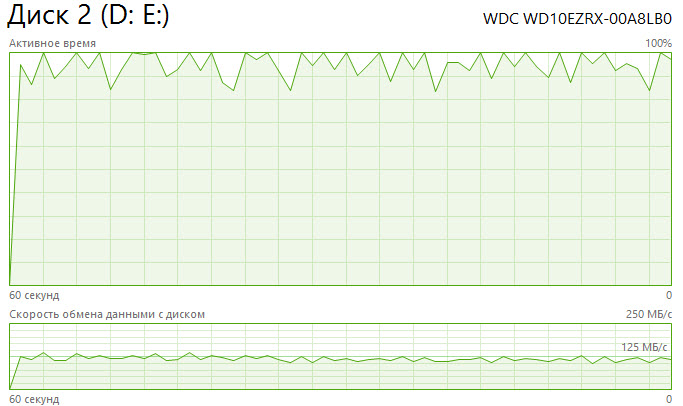
То есть несколько reader`ов просто мешали друг другу.

Время на архивирования файла 1ГБ. При maxActivePartCount=5 и partCount=50 составило 116766 ms.

После этого я отказался от одновременного чтения файла несколькими потоками. ReadersRuner эволюционировал в один Reader. Он не делегировал чтение другим потокам, а выполнял его сам.

После этого графики загрузки стали следующими:





Диск был загружен полностью – единственный reader работал без простоев. Общее время ожидания частей из очереди ForReader стало 281 ms.

Ядра были загружены на 40%. Иногда некоторые arhiver ждали части. Причем не было периодов, когда они все простаивали – нет периодов с загрузкой ядер близкой к 0%. Время ожидания частей из очереди ForArchivers составило 36300 ms.

Время на архивирования файла 1ГБ. При maxActivePartCount=5 и partCount=50 сократилось до 89414 ms.

Некоторые данные замеров в фале «Замеры.xlsx»

## Проверка корректности архивирования

На этапе, когда собственный разархиватор не был готов проверял корректность архивирования следующим образом.

1. Разархивировал архив программой WinRar;
2. У полученного файла и исходного вычислял MD5;
3. Сравнивал MD5;
4. Если совпадали, то архив был «правильный».

## Разархивирование файлов

### Алгоритм разархивирования:

1. Нужно прочитать из архива отдельные заархивированные порции. Они начинаются с 10 байт (31,139,8,0,0,0,0,0,4,0);
2. Эти части по отдельности отдавать на разархивирование (этот этап нужно распараллелить);
3. Полученные разархивированные части записать последовательно в файл.

На 2016.08.29 реализован первый черновой вариант:

* работает только с архивом, состоящим из одной части
* все делает в одном потоке

**Причина:**

Не хватило 7 вечеров. Ушло время на изучение теоретической части и на эксперименты с потоками и способами их синхронизации, так как не было достаточных знаний как правильно их использовать.

## Тесты

Кроме прочих мелких тестов наиболее «важные»:

В проекте ZipLib.Test есть тесты TestCompressDecompress.cs

Тест состоит из:

1. Архивирование;
2. Разархивирование;
3. Сравнение файлов.

Тесты проходят со следующими данными:

* С пустым файлом (EmptyFile.txt)
* С файлом, в котором один байт (data\_1byte.txt)
* С файлом, в котором 10 байт (data\_10byte.txt)

## Планы на следующие версии

Планирую доделать следующее:

1. Доделать разархивирование;
2. Поработать над стратегией разделения файла на части – сейчас она весьма примитивна и сырая.

## PS.

1. Спасибо за интересную задачу. Интересно было попробовать другую предметную область. В плане архитектуры проблем не возникло (рисовать в голове и на бумаге прямоугольники и стрелки приходится очень часто), а вот с потоками и синхронизацией пришлось почитать и пополнить свои знания.
2. В интернете по запросу «gzip распараллеливание» есть немало вопросов «как это сделать?» В том числе натыкался на комментарий, что это задание от Veeam Software. Готовые ответы и решения я не встретил.
3. Если интересна эволюция данного решения (проекта), то исходники в репозитории Git.

https://github.com/stovpyak/gzip.git

Работал в разное время в разных местах – удобно было хранить на сервере… ну и боялся потерять данные. Ветки не использовал. Есть одна тупиковая про Ctrl+C «AbortExit». Но начал и остановился. Решение с отдельным потоком, который слушает нажатия клавиш и при Ctrl+C сообщает основному что нужно прервать работу не заработало. Нажатие клавиш слушало и сообщало что нажато не Ctrl+C, а вот на Ctrl+C приложение просто останавливалось. Видимо на Ctrl+C отрабатывают системные части и выгружают приложение. Как сделать штатное завершение работы не успел разобраться.

Репозиторий открытый - если необходимо, то могу удалить.

# Часть 2 (от 04.09.2016)

30.08.2016 получил ответ со следующими замечаниями и с разрешением продолжить работу над заданием.

1.       Задание не совсем доделано. По сути нет разархивирования (о чем он сам и пишет)

2.       Очень неоптимальное использование памяти. При сборке и запуске в 32-битном приложении падает с OutOfMemory при работе с файлом в 5ГБ. При работе в 64битном занимает в пике 5ГБ

3.       Неоптимальная работа с потоками: на каждую обрабатываемую часть файла создается свой поток

4.       С обработкой исключений тоже проблемы: исключение при чтении файла приводит к падению всего процесса»

## Алгоритм и архитектура.

Пересмотрел набор очередей и их обработчиков. В итоге оставил три очереди и три обработчика очередей.

### Компрессия

Очередь

ForRead

Очередь

ForCompress

Очередь

ForWrite

Reader

Writer

CompressRuner

Compressor

Compressor

1. Reader запрашивает FilePart из очереди ForRead;
2. Читает часть файла заданного размера и записывает прочитанную часть в FilePart.Source, назначает FilePart порядковый номер (index);
3. Передает FilePart далее в очередь ForCompress;
4. CompressRuner получает FilePart из очереди ForCompress;
5. Делегирует компрессию worker`у (в данном случае это Compressor);
6. Compressor запускает компрессию FilePart, беря поток из ThreadPool, результат компрессии помещается в FilePart.Result;
7. По завершении компрессии Compressor передает FilePart в очередь ForWrite;
8. Writer получает FilePart из очереди по порядковому номеру (index);
9. Writer записывает FilePart.Result в архив;
10. Writer передает FilePart в очередь ForRead.

### Плановое завершение работы

Так как Reader знает размер исходного файла, и сколько он уже прочитал частей и какого размера, то он знает, когда файл закончился. Поэтому по завершении чтения файла он у последней FilePart выставляет признак IsLast. И после передачи последней части в следующую очередь завершает своё выполнение.

CompressorRuner так же обработав FilePart с признаком IsLast завершает своё выполнение.

Writer обработав FilePart с признаком IsLast сообщает главному потоку о завершении обработки файла (через событие ManualResetEventSlim \_stopEvent) и завершает своё выполнение.

### Декомпрессия

Архитектура для декомпрессии практически не отличается:

Очередь

ForRead

Очередь

ForDecompress

Очередь

ForWrite

Reader

Writer

DecompressRuner

Compressor

Decompressor

Writer абсолютно тот же, так он записывает FilePart.Result в итоговый файл. В случае компрессии — это будет часть архива, в случае декомпрессии - это будет часть разархивированного файла.

CompressorRuner заменяем на DecompressorRuner, причем в реализации это два потомка одного базового класса с одним перекрытым методом MakeWorker().

Reader тоже удалось сделать один, но он параметризуется разными реализациями PartReader. Для компрессии – это FilePartReader (очень простая реализация – он читает из исходного файла часть определенного размера). Для декомпресии – это ArchivePartReader (он чуть сложнее).

Работа ArchivePartReader:

1. Читает из архива порцию определенного размера;
2. Ищет в ней заголовки 10 байт (31,139,8,0,0,0,0,0,4,0);
3. На основании найденных (или не найденных) заголовков добавляет прочитанную порцию в FilePart;
4. Находя новый заголовок, добавляет все прочитанное до заголовка в текущую FilePart, а заголовок и данные после него оставляет для следующей FilePart.
5. (!) Пришлось учесть, что можно прочитать порцию, которая будет заканчиваться на часть заголовка. В данном случае читаем ещё одну порцию, добавляем к первой и анализируем на предмет наличия заголовков.

## Стратегия компрессии и декомпрессии

Количество одновременно обрабатываемых частей (MaxActivePartCount) выбирается равным количеству процессоров + 30%. Как для компрессии, так и для декомпрессии.

Память, занимаемая этими частями не должна превышать определенного значения. В текущей реализации выбрано 100 МБ (константа).

В итоге размер одной части, которую должен прочитать Reader вычисляется как:

PartSize = 100 МБ / MaxActivePartCount / 2

Делим на 2, так как после архивации в одной части присутствует (пусть и непродолжительное время) и прочитанная часть и обработанная, а мы заранее не знаем степень сжатия - поэтому 2.

Выбирая меньшее значение (не 100МБ), получим бОльшее количество частей файла. Замеры показали что на скорость архивирования это практически не влияет.

## Ответы на замечания

1. Задание не совсем доделано. По сути нет разархивирования (о чем он сам и пишет)

Декомпрессия доделана.

1. Очень неоптимальное использование памяти. При сборке и запуске в 32-битном приложении падает с OutOfMemory при работе с файлом в 5ГБ. При работе в 64битном занимает в пике 5ГБ

Доработан объект SmartCompressStrategy, его работа описана в разделе «Стратегия компрессии и декомпрессии».

1. Неоптимальная работа с потоками: на каждую обрабатываемую часть файла создается свой поток

Изменен способ создания потока – теперь поток получает через ThreadPool.QueueUserWorkItem(Run); Идентификатор полученного из пула потока выводится в отладочной информации.

ThreadId={Thread.CurrentThread.ManagedThreadId}

1. С обработкой исключений тоже проблемы: исключение при чтении файла приводит к падению всего процесса»

Доработана обработка исключений:

Каждому потоку передан обработчик ошибок

Action<Exception> \_exceptionHandler

При возникновении ошибки она обрабатывается в потоке следующим образом

try

{

…

}

catch (Exception ex)

{

\_exceptionHandler(ex);

}

А в самом обработчике ошибки происходит:

* Запоминаем объект exception, переданный из другого потока в \_wasException;
* Вызываем событие \_stopEvent о завершении работы.

После получения сообщения о необходимости завершения работы главный поток:

* Останавливает выполнение «дочерних» (Reader, Writer, CompressRunner)
* Анализирует наличие wasException и в случае его наличия кидает его дальше.
* Далее он обрабатывается уже в главном потоке самим приложением с выводом в лог соответствующей информации.

## Замеры на тестовых данных

Производилась на машине 4-ядра (то есть 5 одновременно обрабатываемых частей).

При этом чтение и запись производились на разные HDD, чтобы отдельно проверить нагрузку на них Reader`ом и Writer`ом.

### Компрессия

|  |  |
| --- | --- |
| **Размер файла** | **Время** |
| 1 ГБ | 10 сек. |
| 10 ГБ | 1 мин 28 сек. |
| 32 ГБ | 6 мин 14 сек. |

Практически всё время было занято Reader`ом. Нагрузка на его HDD была близка к 100%

4 ядра были заняты в среднем на 45-50%

Нагрузки на HDD Write`а практически не было.

### Декомпрессия

|  |  |
| --- | --- |
| **Размер файла** | **Время** |
| 1 ГБ | 4,9 сек. |
| 10 ГБ | 1 мин 13 сек. |
| 32 ГБ | 4 мин 26 сек. |

Практически всё время было занято Writer`ом. Нагрузка на его HDD была близка к 100%

4 ядра были заняты в среднем на 45-50%

Нагрузки на HDD Reader`а практически не было.