0: Other classes

These are classes used by the Read and Write classes used in task 1 and 2.

To measure the time of a function - CodeTimeMeasurer

```
public class CodeTimeMeasurer {

    public static long measureTimeOfFunction(Runnable code) {
        long startTime = System.nanoTime();
        try {
            code.run();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        long endTime = System.nanoTime();
        System.out.println(String.format("Start time of function:
%d\nEnd time of function: %d", startTime, endTime));
        return (endTime - startTime);
    }

}
```

Enums for the blocks - BlockSize

```
public enum BlockSize {

    BLOCK_SIZE(8192),
    N_BLOCKS(131072);

    private int block;

    BlockSize(int block) {
        this.block = block;
    }

    public int getBlock() {
        return block;
    }
}
```

1: Write

After writing to file with 1, 2, ... , 32 GB of data, the results were as follows:

```
The time for 1 GB was 1665 ms, and the throughput is MB/s 629
The time for 2 GB was 2237 ms, and the throughput is MB/s 937
The time for 4 GB was 4865 ms, and the throughput is MB/s 862
The time for 8 GB was 9273 ms, and the throughput is MB/s 904
The time for 16 GB was 18256 ms, and the throughput is MB/s 918
The time for 32 GB was 41343 ms, and the throughput is MB/s 811
```

The system used to generate data was a Macbook Pro from mid 2015, and the data storage system was an SSD (Flash storage). This would explain why the throughput of each iteration was fairly similar for each file write, as a HDD would drop in throughput for increasingly growing file sizes.

The source code for generating 1, 2, …, 32 GB of data to file is supplied below. It is written in Java.

```java
import java.util.*;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.file.*;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

import static java.nio.file.StandardOpenOption.APPEND;
import static java.nio.file.StandardOpenOption.CREATE;

public class Write {
    private static final int[] fileSizes = {1, 2, 4, 8, 16, 32};
    private static List<Long> executedTime = new ArrayList<>();

    private final static int BLOCK_SIZE =
BlockSize.BLOCK_SIZE.getBlock();
    private final static int N_BLOCKS = BlockSize.N_BLOCKS.getBlock();

    private static void writeFile(long fileSize, String filename) throws
IOException {
        Path file = Paths.get(System.getProperty("user.dir"), filename);
        SeekableByteChannel out = Files.newByteChannel(file,
EnumSet.of(CREATE, APPEND));
```

```
        for (int i = 1; i < fileSize * N_BLOCKS ; i++) {
            ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
            out.write(buffer);
        }
    }

    public static void main(String[] args) {
        for (int fileSize : fileSizes) {
            executedTime.add(CodeTimeMeasurer.measureTimeOfFunction(()
-> {
                try {
                    writeFile(fileSize, String.format("filesize_%d_gig",
fileSize));
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }));
        }
        for (int index = 0; index < executedTime.size(); index++) {
            long time = executedTime.get(index);
            long fileSize = (fileSizes[index] * 1024 * 1024);
            long throughput = (fileSize /
TimeUnit.NANOSECONDS.toMillis(time));
            System.out.print(String.format("The time for %d GB was %d
ms, and the write throughput is MB/s %d\n",
                    fileSizes[index],
TimeUnit.NANOSECONDS.toMillis(time), throughput));
        }
    }
}
```

2: Read

The read times for all files from file sizes 1, 2, …, 32 GB, which is significantly faster than the write speeds.

```
The time for 1 GB was 485 ms, and the read throughput is MB/s 2162
The time for 2 GB was 985 ms, and the read throughput is MB/s 2129
The time for 4 GB was 2187 ms, and the read throughput is MB/s 1917
The time for 8 GB was 5033 ms, and the read throughput is MB/s 1666
The time for 16 GB was 10253 ms, and the read throughput is MB/s 1636
The time for 32 GB was 14520 ms, and the read throughput is MB/s 2310
```

The data is generated from the same Mac that was used in task 1. Sources say that the upper limit is around 3500 MB/s (https://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd-speed.html).

The code is similar to the code supplied in task 1, but the buffer is cleared after each read.

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SeekableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.EnumSet;
import java.util.List;
import java.util.concurrent.TimeUnit;

import static java.nio.file.StandardOpenOption.READ;

public class Read {

    private final static int BLOCK_SIZE =
BlockSize.BLOCK_SIZE.getBlock();
    private final static int N_BLOCKS = BlockSize.N_BLOCKS.getBlock();
    private static final int[] fileSizes = {1, 2, 4, 8, 16, 32};
    private static List<Long> executedTime = new ArrayList<>();

    private static void readFile(String filename, int fileSize) throws
IOException {
        Path file = Paths.get(System.getProperty("user.dir"), filename);
        SeekableByteChannel in = Files.newByteChannel(file,
EnumSet.of(READ));
        ByteBuffer buffer = ByteBuffer.allocate(BLOCK_SIZE);
        try {
            for (int i = 0; i < fileSize * N_BLOCKS; i++) {
                in.read(buffer);
                buffer.clear();
            }
            System.out.println(String.format("File %s read", filename));
        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        for (int fileSize : fileSizes) {
            executedTime.add(CodeTimeMeasurer.measureTimeOfFunction(()
-> {
                try {
                    readFile(String.format("filesize_%d_gig", fileSize),
fileSize);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }));
        }

        for (int index = 0; index < executedTime.size(); index++) {
            long time = executedTime.get(index);
            long fileSize = (fileSizes[index] * 1024 * 1024);
            long throughput = (fileSize /
TimeUnit.NANOSECONDS.toMillis(time));
            System.out.print(String.format("The time for %d GB was %d
ms, and the read throughput is MB/s %d\n",
                    fileSizes[index],
TimeUnit.NANOSECONDS.toMillis(time), throughput));
        }
    }

}
```

3: Theory

1) Assume data blocks to be 1KByte. What is the largest file size possible in S5FS?
    a) For data blocks of 1KB, calculating the amount of KBs referred to by the Inode is

       10KB + (1 (triple indirect block) × 256 (double indirect block) × 256 (indirect block) × 256KB (data block)) = 16 777 226KB

       This will amount to 16 777 226KB ≈ 16GB

       The disk contains 13 pointers to disk blocks and the first 10 blocks points each to one block in the file which results in 10 KBs.
       If the file is larger than 10KBs the 11th pointer points to the indirect block which contains up to 256 pointers which results in 256KB.
       Pointer 12 points to a double indirect block which has 256 pointers to indirect blocks and each of them have 256 pointers to data blocks.
       Pointer 13 is a triple indirect block which has 256 pointers to double indirect

block, which has 256 pointers to single indirect block, which has each 256 pointers to data blocks.

However, the file size must be represented as a 32-bit integer, which only allows for about 2GB file sizes.

2) In S5FS creating or deleting a file requires the I-list to be accessed on disk. Why is this not a big problem?
    a) This is not an issue because the I-list is loaded into the primary memory and cached for further use, so when a file has to be created or deleted by accessing its I-node it will be very fast.
3) How does FFS solve the performance problems of S5FS?
    a) FFS solves the problems of S5FS in a few ways:

        Block size: FFS organizes disk blocks by dividing each block into fragments. Disk blocks are larger than in S5FS, and the fragments are kept track of in the file system. This means that a file can occupy only parts of the disk block, so less space is wasted in FFS compared to S5FS.

        Seek time: S5FS is slow due to data blocks being nested into several indirect blocks when the file size is large, which takes longer to read. FFS reduces seek time by organizing the disk into cylinder groups. Related data is kept in close approximation in order to reduce seek time. Inodes are stored in the same data group as the directory containing them, along with the disk blocks that is pointed to. Thus, files in the same directory can be found quickly.

        Rotational latency: S5FS stores data sequentially, which may seem like the logical way, but due to how HDDs read data, this is not the case. After reading a block there is a small delay, and to read the next block requires one whole rotation before it can be accessed. FFS solves this by spacing the data stored with one block spaces, this way HDDs can read the data quicker without having to wait for one whole rotation.
4) Compare soft updates with journaling. What are the advantages and disadvantages with the two approaches?
    a) Journaling: The way journaling works is that steps of transactions are saved into a special journal type file, and when all the steps have been recorded the transaction is written to the file system. This reassures that the steps and transactions are saved. If the system crashes, on reboot, the system can read the journal file to view the unwritten steps.
        i) Advantages: Very quick file recovery.
        ii) Disadvantages: Extra IO actions, as it has to perform logging as well as file system updates.
    b) Soft updates: Keeps a VM as a write system in order to uphold and maintain constraints on block writes. If there is a cyclic constraint on different files, the VM can check for them and lock files during e.g. deletion.
        i) Advantages: Disk structure is always consistent. Recovery of file system is not necessary.

ii) Disadvantages: Resources can end up orphaned (if a directory is deleted, its content might still lie in the file system), however, the integrity is still upheld..

5) Compare extent-based block allocation with cylinder groups. What are the advantages and disadvantages with the two approaches?
    a) Extent-based block allocation, here files are treated as collections of large contiguous regions of the disk.
        i) Advantages: Allows for fast access of large files with a minimum amount of metadata.
        ii) Disadvantages: Fragmentation might be difficult or impossible, because the files (extents) are of variable lengths. If this is inefficient, extent-based block allocation might need as much metadata as group-based systems anyway.
    b) Cylinder groups was mentioned in task 3.
        i) Advantages: Allows for fragmentation, so it is able to access related data quickly (e.g. data in the same directories).
        ii) Disadvantages: About 10% of the disk space is reserved space. Also, even though it is 20 times faster than file systems like S5FS, it is still not very fast.
6) Describe how BTRFS utilizes reference counting to keep track of block reachability in garbage collection.
    a) The garbage collection uses multiple directed acyclic graphs (DAGs), and the trees can reference the same nodes. Due to this, reference counters are being used to measure how many pointers there are to the nodes. The reference counter is not saved into the node, but in a separate tree, which allows for changing the counter without changing the actual node. When the counter reaches 0 the block can be reused.
7) What is the writehole problem? Describe how software RAID-Z as used in ZFS solves the writehole problem.
    a) A write hole issue can arise if there is a power failure on RAID systems, especially RAID5. Avoiding write hole situation can be done by ensuring write atomicity (meaning a file is written completely or not at all). ZFS provides atomicity by utilizing the copy-on-write principle, which is only available on the RAID-Z systems (Not RAID5, for instance). This way, the file system can write to memory first and then do a single write to the file system afterwards.