Masaryk University
Faculty of Informatics

The effects of age on file system performance

Bachelor's Thesis

Samuel Petrovic

Brno, Spring 2017

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Samuel Petrovic

Advisor: Adam Rambousek

Acknowledgement

This is the acknowledgement for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

filesystem, xfs, IO operation, aging, fragmentation ...

Contents

List of Tables

List of Figures

1 Introduction

File systems remain an important part of modern storage solutions. Large, growing databases, multi-media and other storage based applications need to be supported by high-performing infrastructure layer of storing and retrieving information. Such infrastructure have to be provided by operating systems (OS) in form of file system.

Originally, file system was a simple tool developed to handle communication between OS and physical device, but today, it is a very complex piece of software with large set of tools and features to go with. Performance testing is an integral part of development cycle of most of produced software. Because of growing complexity of file systems, performance testing took of as and important part of file system evaluation.

The standard workflow of performance testing of file systems is to run benchmark (e.g. testing tool) on a clean instance of OS and on a clean instance of tested file system [1]. Generally, this workflow present stable and meaningful results, yet, it only gives overall idea of file system behavior in early stage of its life cycle.

File systems, as well as other complex software is subjected to progressive degradation, referred to as software aging [2]. Causes of file system aging are many, but mostly fragmentation of free space, unclustered blocks of data and unreleased memory. This degradation cause problems in performance and functionality over time. Understanding of performance changes of aged file system can help developers in implementing prevention . Such testing fundamentally consists of two steps. First is to bring fresh file system to an aged state and second is the actual performance test of the aged instance.

Foremost, this thesis describe implementation of flexible tests which represent the aforementioned steps. The first test is able to age fresh file system and store the result as an image for later use. The performance statistics collected in the process, as well as resulting layout can be used to evaluate ability of file system to respond to aging. Second test can evaluate resulting image even further by releasing some space and conducting performance test on resulting layout.

Furthermore, using developed tests to test different configurations of file systems and storage is demonstrated. The subject of research are differences between popular Linux file systems (XFS, Ext4) and storage technology (HDD, SSD) in context of aging. Because of nature of collected data, a processing tool was implemented to parse large amount of information into human readable reports. All the generated reports are part of this thesis in form of Appendix A.

The aging tests implemented in this thesis are established as part of testing cycle in Red Hat Kernel Performance team.

In the second chapter, the text present already conducted research of this topic. Third chapter introduce tools used in implementation of tests, while describing their relevant features.

2 Previous research

3 File systems

In this chapter, I present basic information about file systems and describe main features of chosen file system in regard of fragmentation and scalability, which are important topics when discussing file system aging.

File system is a set of tools, methods, logic and structure to control how to store and retreive data on and from device.

The system stores files either continuously or scattered across device. The basic accessed data unit is called a block, which capacity can be set to various sizes. Blocks are labeled as either free or used.

Files which are non-continous are stored in form of extents, which is one or more blocks associated with the file, but stored elsewhere.

Information about how many blocks does a file occupy, as well as other information like date of creation, date of last access or access permissions is known as metadata, e.g. data about stored data. This information is stored separately from the content of files. On modern file systems, metadata are stored in objects called index nodes (e.g. inodes). Each file a file system manages is associated with an inode and every inode has its number in an inode table. On top of that the file system stores metadata about itself (unrelated to any specific file), such as information about bad sectors, free space or block availability in a structure called superblock.

In this thesis, targeted file systems are two most popular Linux file systems, XFS and Ext4 [3], which are also main Red Hat supported file systems. These file systems belong to the group of file systems called journaling file systems.

Journaling file system keeps a structure called journal, which is a buffer of changes not yet commited to the file system. After system failure, these planned changes can be easily read from the journal, thus making the file system easily fully operational, and in correct and consistent state again.

3.1 XFS

XFS [4] is a 64-bit journaling file system known for its high scalability (up to 9 exabytes) and great performance. Such performance is reached by architecture based on allocation groups.

Allocation groups are euqally sized linear regions within file system. Each allocation group manages its own inodes and free space, therefore increasing parallelism. Architecture of this design enables for significant scalability of bandwidth, threads, and size of file system, as well as files, simply because multiple processes and threads can access the file system simultaneously.

XFS allocates space as extents stored in pairs of B+ trees, each pair for each allocation group (improving performance especially when handling large files). One of the B+ trees is indexed by the length of the free extents, while the other is indexed by the starting block of the free extents. This dual indexing scheme allows efficient location of free extents for I/O operations.

Prevention of file system fragmentation consist mainly of a features called delayed allocation and online defragmentation.

Delayed allocation, also called allocate-on-flush is a feature that, when a file is written to the buffer cache, substracts space from the free-space counter, but won't allocate the free-space bitmap. The data is held in memory until it have to be stored because of system call. This approach improves the chance, that the file will be written in a contiguous group of blocks, avoiding fragmentation and reducing CPU usage as well.

3.2 Ext4

Ext4, also called fourth extended filesystem is a 48-bit journaling file system developed as successor of Ext3 for Linux kernel, improving reliability and performance features. Ext4 is scalable up to 1 exbibyte (approx. 1.15 exabyte). Traditional Ext2 and Ext3 block mapping scheme was replaced by extent based approach similar to XFS, which positively affects performance.

Similarly to XFS, Ext4 use delayed allocation to increase performance and reduce fragmentation. For cases of fragmentation that still occur, Ext4 provide support for online defragmentation and e4defrag tool to defragment either single file, or whole file system.

4 Used tools

In this chapter, I presnet tools which were used to implement automated tests for creating and storing aged file systems and measuring their performance. I will describe the main features and means of their usage. All the presented tools are open source projects.

4.1 Beaker

Beaker is a project aimed at automating testing workflow. The software can reserve machine from a pool of labs, install OS and packages, configure environment and perform tasks. The whole process is guided by sequence of instructions in an XML format. Further, I will show few examples of beaker usage.

Example 4.1: Specifying OS to be installed

```
<distroRequires>
  <and>
    <distro_family op="=" value="RedHatEnterpriseLinux7"/>
    <distro_variant op="=" value="Server"/>
    <distro_name op="=" value="RHEL-7.2"/>
    <distro_arch op="=" value="x86_64"/>
  </and>
</distroRequires>
```

Example 4.2: Configuring environment using kickstart

```
<kickstart>
  <![CDATA[
    install
    lang en_US.UTF-8
    skipx
    keyboard us
    rootpw redhat
    firewall --disabled
    authconfig --enableshadow --enablemd5
    selinux --enforcing
    timezone --utc Europe/Prague

    bootloader --location=mbr --driveorder=sda
    zerombr
    clearpart --all --initlabel --drives=sda
    part /boot --fstype=ext2 --size=200 --asprimary --label=BOOT --ondisk=sda
    part /mnt/tests --fstype=ext4 --size=40960 --asprimary --label=MNT --ondisk=sda
    part / --fstype=ext4 --size=1 --grow --asprimary --label=ROOT --ondisk=sda
```

```
    reboot
    %packages --excludedocs --ignoremissing --nobase
    @core
    wget
    python
    dhcpv6-client
    dhclient
    yum
]]>
</kickstart>
```

4.2 FIO

Flexible Input/Output tool is a IO workload generator written by Jens Axboe. It is a tool well known for it's flexibility as well as large group of users and contributors. The flexibility is integral for conductuing less artifical and more natural performance tests. However, approaching more natural test behavior, stability of results drop, so ideal equilibrium between these two parameters has to be found.

FIO accept

4.3 Fs-drift

fs-drift is a very flexible aging test, that can be used to simulate lots of different workloads. The test is based on random file access and randomly generated mix of requests. These requests can be writes, reads, creates, appends, truncates or deletes.

At the beginning of run time, the top directory is empty, and therefore *create* requests success the most, other requests, such as *read* or *delete*, will fail because not many files has yet been created. Over time, as the file system grows, *create* requests began to fail and other requests will more likely succeede. File system will eventually reach a state of equilibrium, when requests are equaly likely to execute. From this point, the file system would not grow anymore, and the test runs unless one of the *STOP* conditions are met (specified with parameters).

The file to perform a request on is randomly chosen from the list of indexes. If the type of random distribution is set to *uniform*, all indexes have the same probability to be chosen, see 4.1. However, if the type of random distribution is set to *gaussian*, the probability will behave according to normal distribution with the center at index 0 and width controled by parameter *gaussian-stddev*. This is usefull for performing cache-tiering tests. Please note, that file index is computed as modulo maximal number of files, therefore instead of accessing negative index values, the test access indexes from the other side of spectrum, see Figure 4.2

Furthermore, fs-drift offers one more option to influence random distribution. After setting parameter *mean-velocity*, fs-drift will choose files by means of moving random distribution. The principle relies on a simulated time, which runs inside the test. For every tick of the simulated time, the center of bell curve will move on the file index array by the value specified using *mean-velocity* parameter. By enabling this feature, the process of testing moves closer to reality by simulating more natural patterns of file system access (the user won't access file system randomly, but rather works with some set of data at a time). On figure Figure 4.3, you can see bell curve moving by 5 units two times.

4.4 Storage generator

Storage generator is a beaker task developed by Jozef Mikovič. It is capable of automated configuration of storage on a machine.

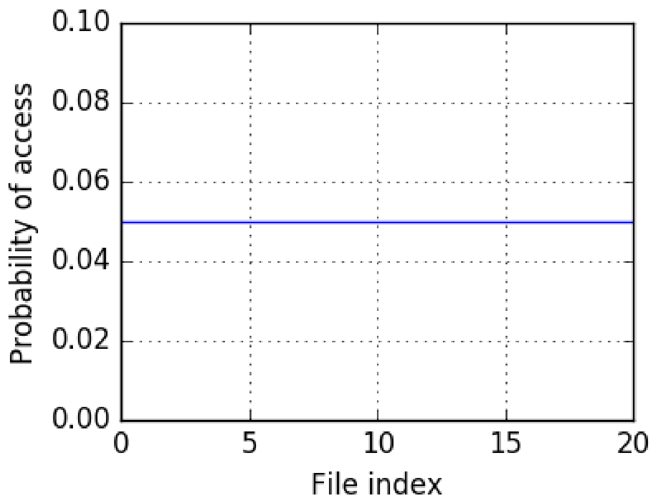

Figure 4.1: Uniform distribution of file access

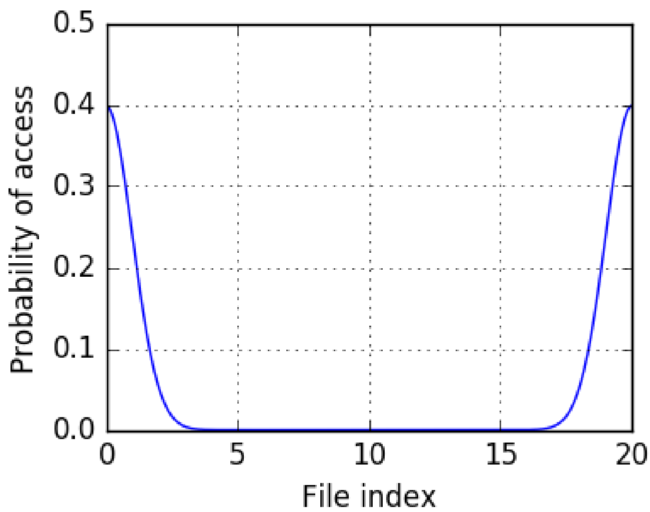

Figure 4.2: Normal distribution of file access

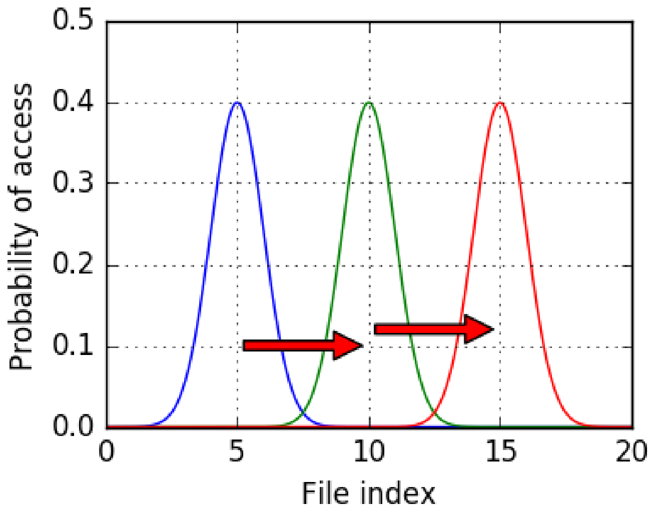

Figure 4.3: Moving random distribution

5 Creating aged file system

5.1 File system images

To achieve consistency of results and to shorten testing time, file system images are used. Once the image is created, it can be stored for later use and replayed back on device. To save space, only metadata of created file system are used, since content of created files is random and therefore irrelevant. Replayed metadata point at various blocks on device, recreating fragmentation while seldom taking significantly less space. These images can be created by using tools developed to inspect file systems in case of emergency. For ext based file systems, there is e2image tool and for xfs, there is xfs_metadump. Both tools create images as sparse files, so compression is needed.

E2image tool can save whole ext based file system or just its metadata and offers compresion of image as well. Created images can be further compressed by tools such bzip2 or tar.

Example 5.1: Creating compressed image using e2image

```
$ e2image -Q $DEVICE $NAME.qcow2
```

Such images can be later replayed back on a device. From that point, file system can be mounted and revised.

Example 5.2: Reloading compressed image

```
$ e2image -r $NAME.qcow2 $DEVICE
```

Xfs_metadump saves XFS file system metadata to a file. Due to privacy reasons file names are obsfucated (can be disabled by -o parameter). As well as e2image tool, the image file is sparse, but xfs_metadump doesn't offer a way to compress the output. However, output can be redirected to stdout from where it can be passed to a compression tool.

Example 5.3: Creating compressed image using xfs_metadump

```
$ xfs_metadump -o $DEVICE -|bzip2 > $NAME
```

Such images, when uncompressed can be replayed back on device by tool xfs_mdrestore. File system can be then mouned and inspected as needed:

Example 5.4: Reloading image using xfs_mdrestore

```
$ xfs_mdrestore $NAME $DEVICE
```

5.2 Implementation

Workflow of image creating is contained in the package drift_job. After extracting fs-drift, the main script starts python script, which handles the process of running fs-drift. Settings of fs-drift are passed as a parameter and are parsed inside the script. Before running the fs-drift, python daemon thread is created to log free space fragmentation periodically while fs-drift is running. After the aging process is done, overall fragmentation is computed.

After the aging process, the script use system tools to create and compress the image. Information about system is gathered as well and all the logs are archived and sent to data collecting server. Parameters available for drift_job:

1. -s | –sync, flag to signalise wheather or not to send data to server (usefull for developing purposes)
2. -m | –mountpoint
3. -d | –device
4. -r | –recipe, parameters to pass to fs-drift
5. -t | –tag, string to distinguish different tests

5.3 Fs-drift/aging settings

As the creator states in README, to fill up a file system, maximum number of files and mean size of file should be defined such that the product is greater than the available space. So if the workload is supposed to fill 500GB of space, while having maximum file size of 1GB (therefore mean size is 500MB), maximum number of files should be much higher than 1000. Optimal approach is to define seemingly no upper limit to let the fs-drift fill the volume, therefore numbers as high as 10^8.

Parameter -t specifies the top directory, which will be used in test, in this workflow it is set to $MOUNTPOINT.

There is an option to specifiy user-defined file to use as a workload table, which is a desired percentual representation of operations in a workload. Since the goal of this workload is to create fragmented file system in a short time, read and rename operations are irrelevant. Therefore only create, append and delete have representation in this workload. The optimal results were reached when every operation had equal representation, e.g. 33%

The fs-drift allows directories up to defined level to create. The directory in which a file is directly affect its chance to be selected for a chosen operation, so by using only one directory, the equilibrium happens too fast, long before the file system is filled completely. Therefore we allow up to three levels of directories to be created.

Duration of the test is set to 5 hours so the test is usable for testing campaign without oversaturating of the servers.

6 Performance testing of aged file system

6.1 Test structure

Performance testing of created images is done by a package recipe_fio_aging. Upon instalation of necessary tools (libs, fio), the package finds and downloads coresponding file system image according to obtained parameters. As shown, images are stored compressed, therefore decompression is needed after download. Once these steps are succesfully completed, the image is restored on the device by using presented tools (e2image, xfs_mdrestore). If the image restoring completes succesfully, file system can be mounted and worked with exactly like it would be just after the aging process.

After image restoration, some amount of the files needs to be deleted to create space for the performance test to take place. The files to be removed are chosen randomly until desired amount of volume has been freed. By using this workflow, e.g. freeing some amount of space, we can simulate aged file system in various phases of aging by using just one image of a very fragmented file system.

When free space is reclaimed, FIO test will take place using parameters given to recipe_fio_aging. The overall space occupied by the test should not be larger than available space on the file system, otherwise the test will either fail completely or report incorrect results.

For statistical correctness, the FIO test can run several times in a row. After last iteration, the results are archived and sent to data-collecting server.

Parameters available for recipe_fio_aging:

1. -s | –sync, flag to signalise wheather or not to send data to server (usefull for developing purposes)
2. -n | –numjobs, number of test repetitions. For statistical stability
3. -m | –mountpoint
4. -d | –device
5. -r | –recipe, parameters to pass to FIO test
6. -t | –tag, string to distinguish different tests

6.2 Benchmark settings

To ensure stability of test results, I decided to use simple form of standard performance test.

7 Testing environment

7.1 Storage

HDD is a rotational disk, which requires specific approach from kernel, to ensure the lowest possible seek time. Seek time is a time for moving parts of the device to find next relevant block of data. This affect overall performance greatly, because with large fragmentation, seek time becomes quite high.

As for SSD, this type of device does not have any moving parts, which make perform really well. One of the problems, however, is limited lifecycle of memory cells. SSD manufacturers deal with this problem by adding controler with its own scheduler, which make sure, no parts of the device are used significantly more than other parts.

When aging the filesystems, I expect for those grown on HDD to perform significantly slower after aging process, and I expect SSD filesystems not to be affected at all, or maybe significantly less.

7.1.1 HDD

7.1.2 SSD

7.2 Used machines

The aging process took place on these Machines:

1. Model: LENOVO System x3250 M6
2. CPU: Intel(R) Xeon(R) CPU E3-1230 v5 @ 3.40GHz (4 cores), arch i386 x86_64
3. Memory: 16384 MB
4. Storage:
   (a) EG0600FBVFP HP Proliant HardDrive
   (b) Interface: Serial Attached SCSI
   (c) Capacity: 600 GB

1. Model: IBM x3650 M4
2. CPU: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
3. Memory: 65536 MB
4. Storage:
   (a) 2xSSDSC2BB480G4i IBM Solid State Drive

(b) Interface: Serial ATA
(c) Capacity: 480 GB

The system installed on machines is RHEL-7.2 with kernel 3.10.0-514.el7.x86_64

8 Results

The output of result generator is a htlm report summarising all information about system, links to raw data and charts of measured values.

8.1 Performance of aged file system

8.2 Differences betweem XFS and EXT4

8.3 Differences accross different storage

9 Conclusion

Here I will admit, that these results were not really surprising and ABSOLUTELY no breakthrough, however, as noone really research this branch of QE, the results are definitely a step further in this field.

Bibliography

[1] A. Traeger et al. "A Nine Year Study of File System and Storage Benchmarking". In: *ACM Transactions on Storage (TOS)* 4.2 (2008), pp. 25–80.

[2] Domenico Cotroneo et al. "Software Aging Analysis of the Linux Operating System". In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ISSRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 71–80. ISBN: 978-0-7695-4255-3. DOI: 10.1109/ISSRE.2010.24. URL: http://dx.doi.org/10.1109/ISSRE.2010.24.

[3] Lanyue Lu et al. "A Study of Linux File System Evolution". In: *Trans. Storage* 10.1 (Jan. 2014), 3:1–3:32. ISSN: 1553-3077. DOI: 10.1145/2560012. URL: http://doi.acm.org/10.1145/2560012.

[4] *Project XFS Linux*. 1999. URL: http://oss.sgi.com/projects/xfs/ (visited on 04/20/2017).