CS574 OPERATING SYSTEM, PROJECT 3 - FILE SYSTEM MEASUREMENTS

Sajidur Rahman

Source code: https://github.com/Reznov9185/file_system_tests)

1. ABSTRACT

The goal of this work is to understand UNIX file systems and study the nature of its behaviors as well as have a solid understanding of why and how drove the design decisions of designing a robust operating system. Here we tried to run some tests and try to analyze the data from it to come towards some "empirical proofs" regarding our chosen UNIX-based system. For measurements, we tested and used CPU-cycle counter side by side with the normal timers to time the experimental operations. With this study, I started with the block size of the file system and studied their size varying the size to figure out whether our reported block makes sense or not considering the prefetching effect. Moving forward I tried understanding the prefetching effect on the large files and found that they are mostly a bit random rather than some clear outliers. I figure out a UNIX tool for getting the file cache size and then went to experimenting with that size and found that the reads out of that block are mostly more expensive than the reading in that size. Lastly, the focus was to figure out the file allocation strategy and get, an understanding of the numbers of the direct inode-pointers for the indirect block mapping scheme and extend-size for the extend-based schemes.

2. INTRODUCTION

In this study, I have learned and understood the inner file system's workings behind a robust operating system. Found the concept of these engineering choices interesting and fun. The block size, prefetching, different aspect of different caching, and inode-based file allocation strategy was major learning of this work for me. I found that where some things are very straightforward to read in theory, it's very hard to prove in practice and in real-time with real-world systems as they depend on a lot of other variables as well.

3. METHODOLOGY

The first thing to decide on in this study was choosing the operating system and I have chosen my work machine which is running a 64-bit Ubuntu 20.04.4LTS, Kernel is Linux version 5.13.0-40-generic running on top of an 8 core, Intel® Core™ i5-8250U CPU @1.60GHz. I also tested my work on the Linux-based systems in our labs remotely.

For approaching the problems, like any other measurement experiment, the question starts with the tool to measure this. My tool of choice was rdtsc() function which is a CPU-cycle counter and can measure the change of cycle and I verified it by comparison with the regular method of gettimeofday() using sleep(10) to verify that they are similar in measurements in big scale but from my experiments, rdtsc() can explore some things that are hard to explain only with time in measurements.

As for the tools, I have used c++ as my language to do these types of syscalls() effectively with the help of the platform APIs and for visualization of the data, I have used matplotlib on python3. As I will discuss the result that I have found from my systems, but to reproduce it the reader can use the readme file from the repository that I have included with this paper.

In the experiments the strategy was to, start with simple measurements and then explore the idea of testing the system with varying parameters and storing them in the files for the data visualizations. The first step is to reason with the ground truth which can be found from platform tools and later move the parameters to play around and compare with the ground truth to get closer to the results.

4. RESULTS

To summarize the findings, I have found that the block size is 4096 bytes or 4 kilobytes(KB), I have got that from the stat() call as well as from the experiments which I will discuss with the findings of the experiment. I have not found any pattern for the prefetching though and the prefetching seems to be occurring as randomly as it could or there is some underlying strategy that I missed. For large files like 10 gigabytes(GB) and for its sequential reads could find any interesting patterns, I also repeated this experiment with different large file sizes. For the file cache size, I have found that it is 64 bytes which are found via L1 cache line size, as confirmed by the experiments it is found that in the most of those runs if it's in the line size it takes less average time which is shown as average dotted lines on the plots. The limitation here is ensuring the underlying cache is as it is hard to assure that it is within that size for sure. In the file system's allocation strategy it is found that mine is an extension-based scheme that is in the ext4 format. The extent size is 16 megabytes(MB), and these

are confirmed by the syscalls() and lsblk. When experimented with the size makes 2 very prominent clusters of data plot of the read() times on the different sizes like the one showed later from 8MB to 24MB.

In Fig. 1, it is a read time in second varying file sizes from 1 to 32KB. Though it is found that some initial read takes a lot of time are thins are all over the place we can certainly see that there are 4-5 points that can be grouped by time, as they are the 4KB blocks. These 4KB blocks are also very different in the comparison of average cycle variance which can be found by running things from the source code.

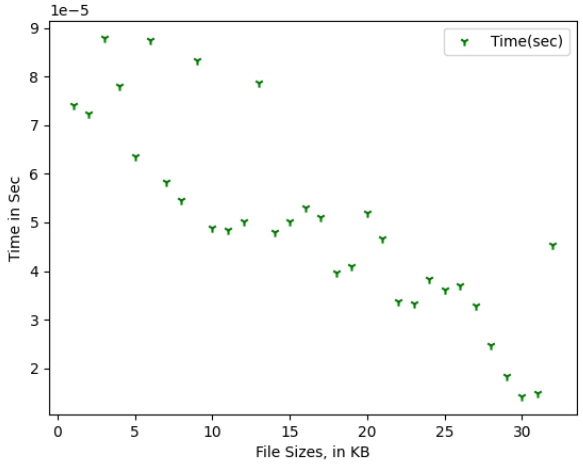

Fig. 1. Block wise varying size(KB) read times. 4KB Clusters are found

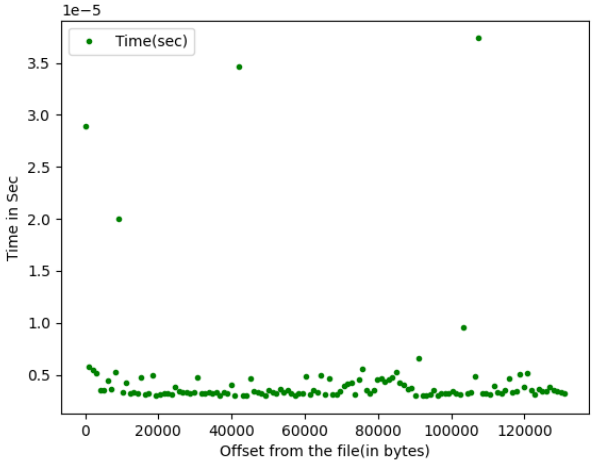

Fig. 2. Prefetch detection in reading large file.(10GB)

It is a sequential read() test on 10GB data with different offsets to read from to analyze the prefetching pattern for Fig. 2. This remains inconclusive as it is quite random in nature and in other experiments with different sizes as well, except for some obvious outliers.

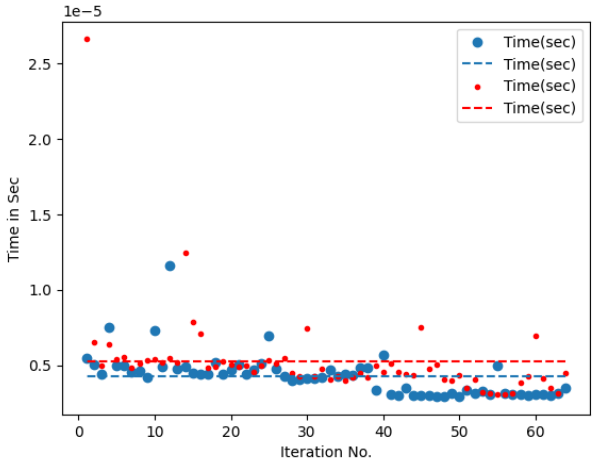

Fig. 3. Read same file cache block vs seperate ones

For Fig. 3, with different iterations, it is compared as the in cache access vs out of cache access and compared their averages with the dotted lines. As it is found, in most runs of this experiment with separate settings of parameters as well, the in cache 64byte line has quicker overall read() times.

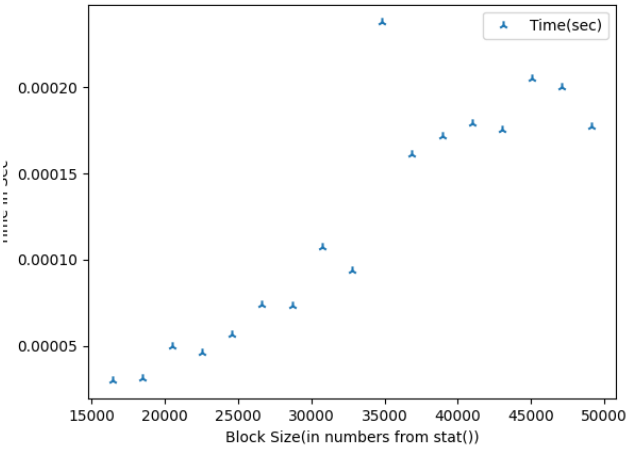

Fig. 4. Timing based on block size (8MB to 24MB's block sizes)

As the extent-based allocation scheme is found in my system, for the extended size of 16 MB it is tried to plot the read() times around the 16MB block, and interestingly in this incremental nature of this plotting in Fig. 4, it is found that there is a prominent distinctive zone between them.

About the data and the experiments, I have found that with different parameters and variables it seemed like they

are quite consistent, except the thing in Fig.2 as it is quite inconclusive which could be something I might be missing in that experiment's context.

5. CONCLUSIONS

The block size is found to be 4096KB, prefetching causes some outliers. but they can be found with time variances and cycle variances. For big files, the prefetching does not show any pattern for the current experimental settings. Read accesses within the file cache block size, on the average case is faster that the once way out of that file line cache. For the last experiment, extent-based schemes of extent size of 16MB show a distinctive zone in the read times. In summary, these experiments explored the strategy of a UNIX file system design.