Atiya Kailany
December 9, 2019
CS 474 – Operating Systems

Project 2: File System Measurements
Author: Atiya Kailany

Abstract

For this project we are looking at different behaviors in operating system functions such as read(). While performing small file system commands, the executed code snippets are surrounded by a CPU cycle counter known as rdtsc() and the obtained time is used to deduce some conclusions about four main questions. The questions are:

- How big is the block size used by the file system to read data?
- During a sequential read of a large file, how much data is prefetched by the file system?
- How big is the file cache?
- What is the allocation method for the file system? For file systems using indirect block mapping scheme, how many direct pointers are in the inode? For systems using extent- based scheme, how big is the extent size?

The obtained time is then plotted into graphs, and these graphs are then used to determine the solutions for each question based on the Linux system that is being tested.

Introduction

In this project, we are exposed to some memory mapping and memory allocation behavior of the Linux system being tested. Each question is designed to allow the researcher to explore an aspect of operating systems and their behavior. For example, it has been made clear through the series of experiments performed, that when executing using cache as memory the results are

significantly faster compared to when using disk space. Other patterns emerge such as the allocation method for the file system, which in the case of the Linux operating system used in this lab was extent-scheme. Another outcome was that when a cache fills up after a series of commands, it start allocating memory from the disk, and when the fetching happens on the disk, it harms the execution speed significantly. Finally, all these conclusions were drawn from the graphs plotted for each experiment, especially after successfully determining the block size of 4096 bytes.

Methodology

For each of the questions asked in this project, a different experiment was used to determine the size or the required measurement.

Block size:

For the block size, a total of 100 reads of varying sizes were performed (256, 512..etc). However, the best and most clear plot was displayed when plotting for block size with read size of 256 bytes. So the strategy was to read with a particular size a 100 times in a for loop, and I would surround the read() command with CPU cycle capture then compute the elapsed time and convert it to microseconds. This experiment worked well, as it helped show a number of spikes in time that helped determine the block size of 4096 bytes.

Prefetched Data Size:

For this dilemma, a similar experiment was performed. Again, the read command would be measured, and the elapsed time has to be computed. However, the read command now reads about 4096 bytes at a time since we determined the block size in the previous experiment. The reads should stay at a constant speed until we see a spike after a certain number of reads. The moment where the spike occurs is the cut-off point of the number of blocks that signify the prefetch data size. For example, if the spike occurred after 15 iterations, then the size of prefetched data is 15 blocks of 4096 bytes.

File Cache Size:

For this experiment, a total of 50 iterations of reads were performed and the time was recorded for each read. Around iteration 14 the time spiked which signifies that the main memory has been filled and now the operating system has to access the disk to obtain the required data for the read system calls. The number of reads that the cache was able to process before the time spiked, signifies the caches size of the file system.

Allocation method: extent-scheme

Through some research the allocation method for the computer science department machines was determined to be extent scheme based. Therefore, an experiment for the number of inodes was deemed inappropriate. To determine the extent size, the system calls write() and fsync() were used. The code snippet would write a total of 1 megabyte to a file and then sync the changes and measure the time it took for each sync. Using the same algorithm, the amount of iterations of syncs before the graph spiked would signify the extent size. For this experiment, the graph spiked around iteration 85.

Results

For each of the above-mentioned experiments, a plot was graphed to determine the required information. See results below.

Block size:

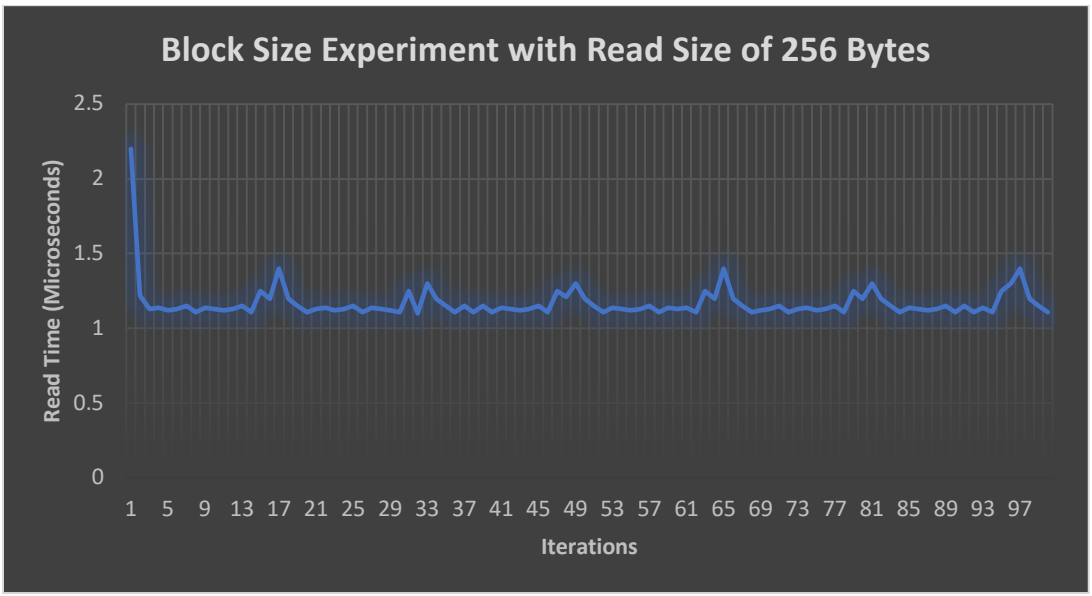
The following code snippet would read 256 bytes per iteration and record the time it took for each iteration's read and print it on the terminal. This was performed for 100 iterations and the graph below was made using the iterations as an X-axis and time it took as a Y-axis.

```
for(i=0; i<100; i++) {
  start = rdtsc();
  size = read(fd, buff, _256_);
  end = rdtsc();

  timeSpent = (((double)(end - start) / conversion) * 1000000);

  printf("Iteration %d", i+1);
  printf(" Block Size: %f\n", timeSpent);
}//for
```

Graph plotting the block size experiment over number of iterations and using reads of 256 bytes:



As seen by the above graphs, the read time shows spikes multiple times throughout the iterations. When inspected, the spikes occur at 1, 17, 33, 49, 65, 81, and 97 this shows a pattern, which is that they occur every 16 iterations. Since this is a 256-byte size read, using these two pieces of information we can compute the block size. The block size is equal to 16 * 256 bytes which is 4096 bytes. Therefore, the block size for this file system is 4096 bytes.

Prefetched Data Size:

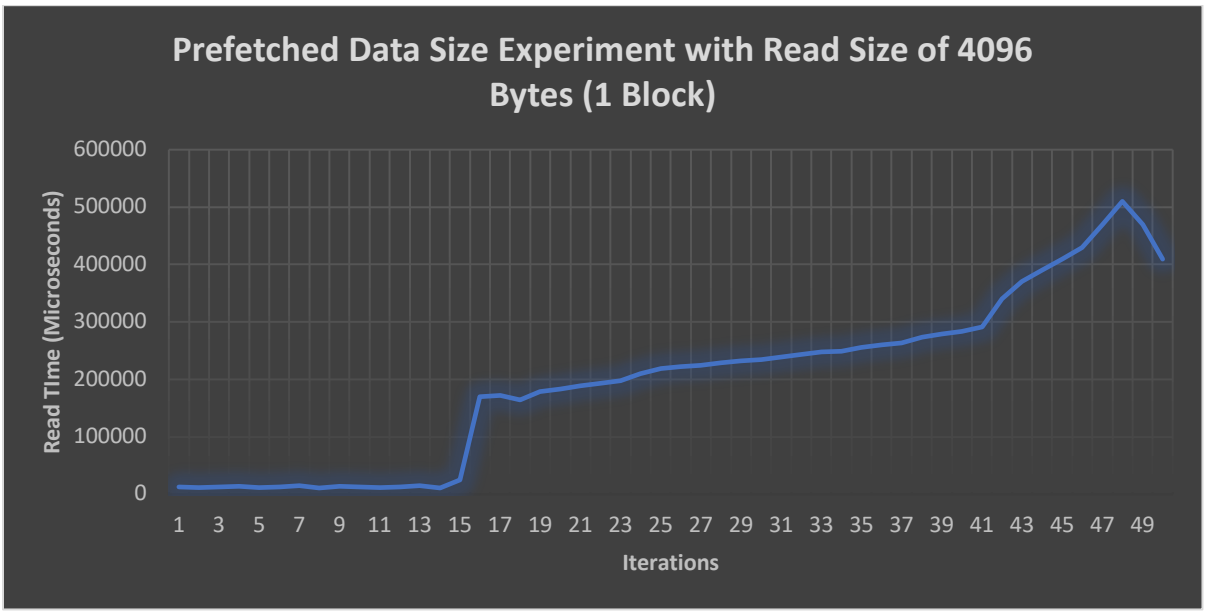
For this experiment, since we know the block size, we will be reading 4096 bytes at a time. This means we will read one block at a time and measure the time it took, when the time it takes for each read starts to spike that means we found the cut-off point of the prefetched data size.

```
for(i=0; i<50; i++) {
  start = rdtsc();
  size = read(fd, buff, _4096_);
  end = rdtsc();

  timeSpent = (((double)(end - start) / conversion) * 1000000);

  printf("Iteration %d", i+1);
  printf(" Prefetched Data Size: %f\n", timeSpent);
}//for
```

Graph plotting the prefetched data size experiment and read time for each block:



As shown above, the read of each block was consistent until we see a spike around iteration 16 and it the graph keeps on getting higher. Using this graph, we can deduce that the system was able to prefetch around 15 blocks of data before running out of space. Therefore, we conclude that the file system has a prefetch size of 15 blocks of 4096 bytes.

File Cache Size:

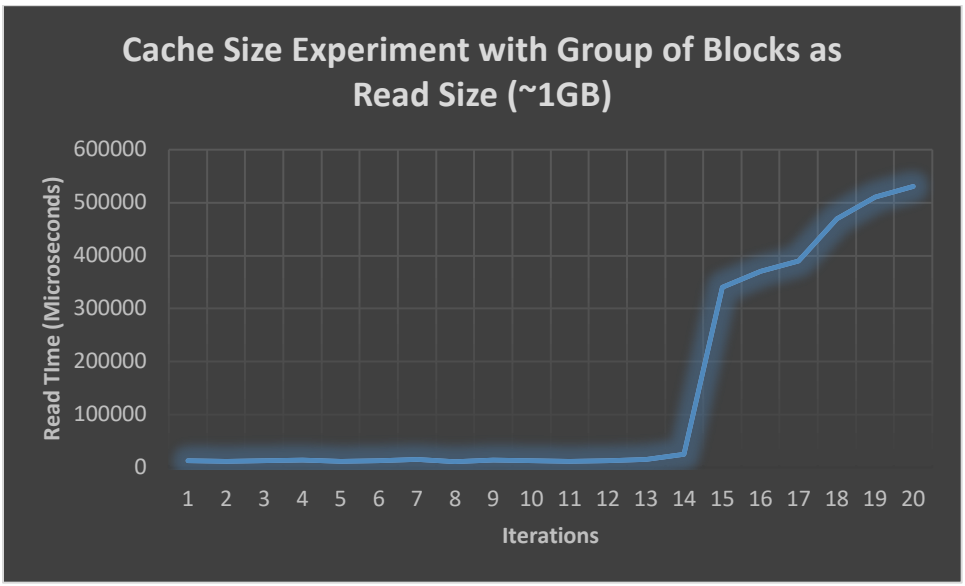
For this experiment, the file cache size is usually pretty large, thus making reads of 4096 bytes will not be able to do the job with a minimal number of iterations. Therefore, I assumed that at least the cache size has to be 1gigabyte. And due to this assumption, the following code snippet is actually reading a group of blocks, which equals to 1 gigabyte of data, at a time.

```
//loop 20 times so we can fill the memory and see where the spike occurs
for(i=0; i<20; i++) {
    start = rdtsc();
    size = read(fd, buff, _1GB_);
    end = rdtsc();


    timeSpent = (((double)(end - start) / conversion) * 1000000);

    printf("Iteration %d", i+1);
    printf(" Cache Size: %f\n", timeSpent);
}//for
```

Graph plotting the Read time for a group of blocks over 20 iterations:



As seen in the above graph, the reading of the group of blocks had a low consistent time, until it reached iteration 14 where it had a significant spike occur. This means the file system cache was able to read 13 iterations before running out of memory. These 13 iterations equate to around 13 gigabytes of size, since each read of the group of blocks is around 1 gigabyte. Therefore, the file system cache size is approximately 13 gigabytes.

Extent size:

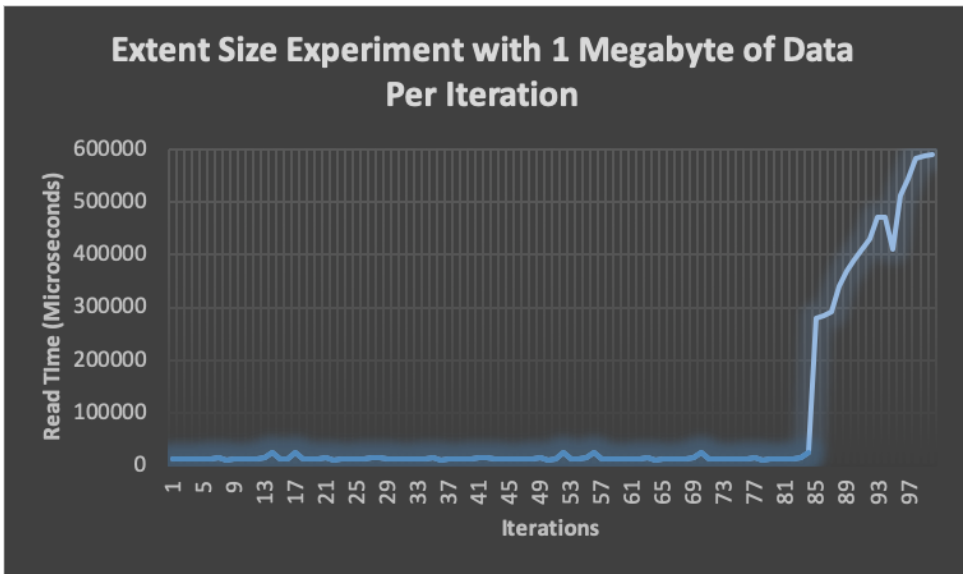
The extent size for this experiment was determined through using the system function calls write() and fsync(). The code snippets below, would write a total of 1 megabyte to the destination file then sync it. The time it takes to execute the sync command is recorded, this process is executed in 100 iterations.

```
for(i=0; i<100; i++) {
  //writing 1 megabyte of data
    size = write(fd, buff, _1MG_);
    start = rdtsc();
    size = fsync(fd);
    end = rdtsc();

    timeSpent = (((double)(end - start) / conversion) * 1000000);

    printf("Iteration %d", i+1);
    printf(" Extent Size: %f\n", timeSpent);
  }//end for
```

Graph plotting the sync time of 1-megabyte data size per iteration:



Each iteration in the above graph syncs in 1 megabyte of data per iteration, using this algorithm we can determine the Extent size by monitoring any spike in the graph at a certain iteration. As seen above, the graph starts to spike after 84 iterations, which means that 84 megabytes of data were synced before running out of space. Therefore, the extent size for this file system is approximately 84 megabytes.

Conclusion

In conclusion this experiment exposed me to a lot of different aspects about the operating system and its behavior. Especially file systems and how run time speed is affected by the sizes of data being processed. The experiments performed in this project helped me find a series of discoveries. First the algorithm that the system used to process data, and how memory is allocated. Second, the prefetch and cache size experiments helped me establish a solid understanding of the relationship between cache, RAM and hard-disk. Finally, this project helped me learn how the operating system functions as a whole, and the decision making and back-end process behind every script command. It really goes to show how complicated and delicate these systems are. As well as the dedication the creators of these systems had to providing optimum performance at the lowest cost possible.