File System Measurements

Authors: John Gangemi & Raj Patel

Abstract:

This paper tries to uncover the specifics to a Unix based file system by timing file system calls with a highly accurate hardware based cycle counter known as rdtsc on the intel x86 micro-architecture. Measurements acquired from extensive tests prove the complexity of the file system.

Introduction:

What we are concerned with in this paper is determining some of the inner workings of the file system of a Unix based machine with ext2/ext3 format. In our case we used a Unix based machine in the University of South Florida C4 lab (more specifically C4 machine 09). We begin the experiment by determining a suitable timer to measure the file system. The two timers that we chose from are gettimeofday() and rdstc(). We did a test to see which is better and ultimately decided to go with rdtsc() due to its simplicity. Afterwards we began the measurements for the file system starting with the size of the block. Using stat-f command in the terminal we were able to determine that the blocksize is 4KB (4096 bytes) and replicate this in code using our own experiment. Next we tried to determine the prefetch buffer size by timing reads of n blocks, sequentially for a large file. Once we have calculated the prefetch size we can make an assumption for the number of blocks for the file cache, using a method described in further detail in the methodology section. Lastly we attempted to figure out the number of direct inode pointers by doing a number of sequential writes to sequential blocks and seeing when the time per write increased.

Through timing alone we were not able to figure out the file system characteristics but we do believe that the solutions are logically correct. The file system is very complex and it is often times hard to see what is going on behind the scene since there are many levels of abstractions that are taking place.

Methodology:

Timing

The first task that we tackled was determining a suitable timer since the accuracy of the timer is critical when doing measurements with the file system. The two timers that we compared were rdtsc() and gettimeofday().

Rdtsc() returns a data type uint64_t which is defined in the stdint.h header file so in order to obtain meaningful data from this timer we had to do some calculations (which are shown below). One of the things we had to determine to use the rdtsc() as a timer was the clock frequency of the machine. We determined the frequency by typing “cat /proc/cpuinfo” in the terminal. This command told us the frequency of the machine and also some other useful information such as the L3 cache size which was 8192 KB. The frequency of our specific machine (C4 machine 9) was 3393 MHz.

Calculations for timer in nanoseconds using rdtsc()

Elapsed clock cycles/CPU frequency = seconds
Nanoseconds = seconds x 1E9

According to initial research online and advice given to us by our professor rdtsc() is more accurate. However it seemed that when we ran our experiment using the system call sleep() and putting the machine to sleep initially for 2 seconds, gettimeofday() was slightly closer to the actual value of 2. Thinking this might be an anomaly we decided to run sleep with different amounts of times which varied from 10 seconds to 4 minutes but still each time gettimeofday() was closer to the actual value. It should be noted that the times we were comparing were in nanoseconds, and that gettimeofday() returns microseconds so it was converted to nanoseconds. Ultimately we decided to use rdtsc() for timing since it was simpler to implement and the numbers were not that different from the results given to us by gettimeofday().

Determining Blocksize

After determining a suitable timer we decided to measure the file system blocksize by reading random bytes of data of increasing magnitude. Doing some initial research we were able to discover "stat" which is a quick way of finding relevant information about a file. When typing stat followed by the file name into the terminal we were able to see relevant information which included: number of bytes, number of blocks (which is the number of disk sectors) and the I/O block size which was 4096 (it seems to correlate to the file system block size). Afterwards using the man pages on stat we came across "stat –f" which gave more information than stat did. When typed into the terminal followed by the file name we were able to determine that the block size was indeed 4096 bytes(4KB). Ultimately we were able to find a way of actually doing the equivalent in the c code itself which is shown below:

Struct statvfs fs;
//open random file
Statvfs (fd,&fs);
Blocksize = fs.f_bsize;

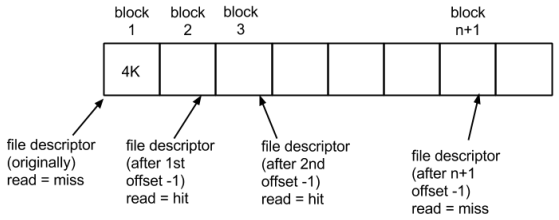
Through research and the above stated methods we were able to determine the block size is 4KB, however we wanted to prove it through experimentation. In order to do this we created an array of size 7 which included the following bytes [512, 128, 1024, 16, 2048, 256, 4096]. These bytes were all read from a 4K file that we created. We hypothesize that for the first read of 512 bytes will be a miss because the file cache does not contain the relevant block so the cost of the read will be large since you have to go to disk to retrieve said block. After that point the rest of the reads will be less except for 4096 which should be another miss and therefore have a time close to the initial miss. The results are showcased in the results section of the report.

Determining Prefetching Buffer Size

We tried two methods to determine the prefetching buffer size however both attempts while sound in logic did not give us the results that we expected.

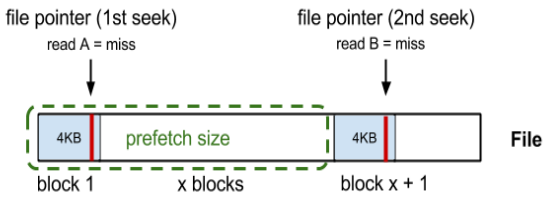
The first method involved opening a large file on our local file system and reading for n blocks (which we believed to be the size of the prefetch buffer). Our main loop used lseek to increment the pointer of the file descriptor using an offset equal to the file system block size minus one. After seeking to the correct location in the file we then read a single byte of data in the respective block and timed how long it took. We then repeated the process for n +1 blocks which is what we believed was one more than the size of the prefetch buffer. The timing for the n+1 block should be equal to the time it took to read from the first block i.e. a cache miss.

Diagram of first method



The second method relies heavily on the assumption that we know the approximate prefetch buffer size. Again we open a large file and seek to the beginning of the file using lseek and read a byte of data from the first block thus ensuring a cache miss and subsequently a prefetching of x number of blocks. X blocks is the determining factor for the next lseek operation for which we then read a byte of data in the block that is x blocks away from the first . By reading the current block pointed to by the file descriptor. We can compare it's time to the first block for which they both should be equal therefore implying that you have reached the second cache miss and consequently proved the size of the prefetch buffer.

Diagram of second method



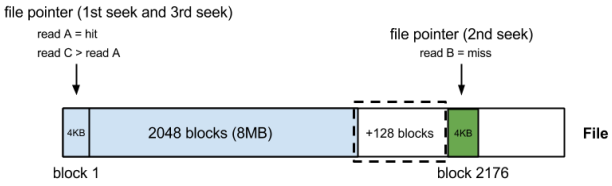
Determining Size of the file Cache

The method that we used to determine the size of the file cache is as follows. We read from a large file, 8 MB (L3 cache size of the intel i7-2600) of data to fill the cache. Then read block one and timed that. Next we read any block beyond the assumed cache size plus the prefetch size and timed it (this should be a relatively large value equal to that of a cache miss). Finally we read and timed the first block again since the first block should have been replaced in the cache at this point (the time once again should be relatively large as a result).

Calculations used for determining number of blocks in cache

Cache = 8192 KB, Blocksize = 4KB
Approx. blocks for cache = cache/blocksize = 2048

Diagram showcasing method used to determine file cache



Determining direct pointers in the inode

We created a 1 MB file and then using an arbitrary number of writes to the file in a sequential fashion we wrote a byte to each block using the file pointer of the file descriptor where every iteration of the loop the file pointer was offset by the size of a block (4 KB). Timing each write should show the number of blocks at the beginning of the file that have direct pointers because those write times will be significantly smaller than preceding blocks. We know this because indirect pointers point to a block of data which holds pointers to other blocks which contain the desired data. Therefore the write time for indirect pointers should be nearly double that of a direct pointer.

Fsync was used to ensure that writes will happen that instance and not be sent to a buffer for later I/O operations.

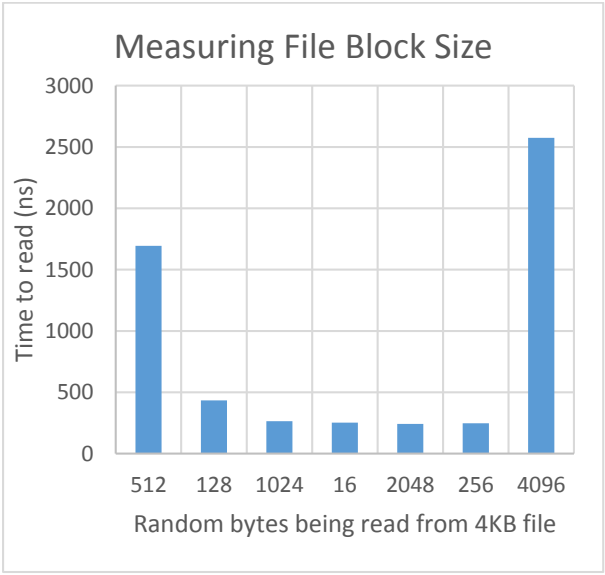
Results:

Determining Blocksize

Output in the terminal when the function FileBlockSize() was run

```
Actual file system block size: 4096 bytes (4KB)
Byte 512 read time: 1652.814618 ns
Byte 128 read time: 416.150899 ns
Byte 1024 read time: 288.829944 ns
Byte 16 read time: 264.073092 ns
Byte 2048 read time: 264.073092 ns
Byte 256 read time: 253.463012 ns
Byte 4096 read time: 2424.992632 ns
```

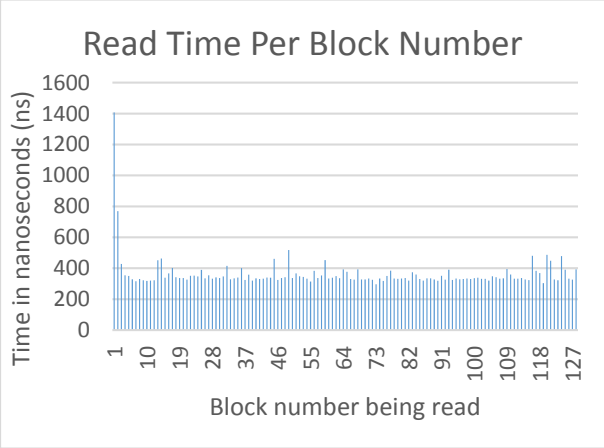
Graphical representation of the output of FileBlockSize() function



It is noted here that the first read to the file incurred a cache miss penalty of approximately 1700 nanoseconds. This is inline with the cache being empty upon an initial read. Subsequent reads are all within the same same block therefore they are all in the cache. However when reading byte 4096 you try to access a block that is not in cache thus a cache miss penalty.

Determining Prefetching

Graphical representation of the output of PrefetchingSize() function



Results show an initial read to the file incurred a cache miss penalty and no correlation between a cache miss penalty and subsequent spikes in the graph can be made. We would have expected to see large spikes dispersed evenly throughout the graph or a single spike at 128 (due to sources online)

Determining Size of the file Cache

Output in the terminal when the function FileCacheSize() was run

```
Measuring File Cache Size . . .
Block 1 read time: 417.329797 ns
Block 2176 (cache + prefetch) read time: 562.334218 ns
Block 1 read time: 265.251989 ns
```

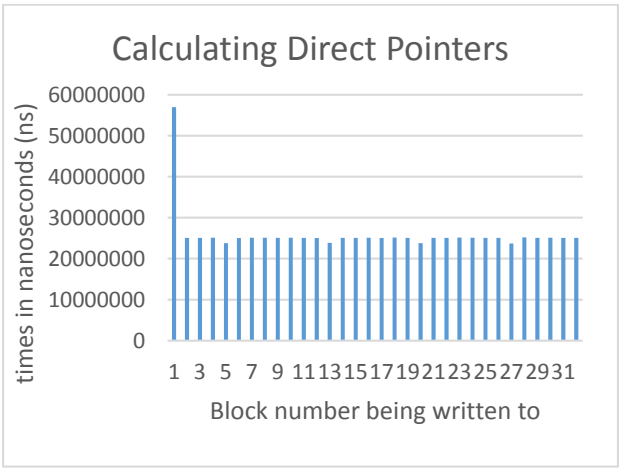
Results indicate are assumption of the cache behaving like a queue where the first block would have been removed after reading block 2176 is in fact incorrect and we now believe that the cache implements a least recently used scheduling policy.

Determining direct pointers in the inode

Output in the terminal when the function DirectPointers() was run

```
Calculating Number Of Direct Pointers . . .
Block 1 write time: 56987739.463602 ns
Block 2 write time: 25081326.849396 ns
Block 3 write time: 25082935.455349 ns
Block 4 write time: 25095932.213380 ns
Block 5 write time: 23790582.964928 ns
Block 6 write time: 25020837.606838 ns
Block 7 write time: 25128741.526673 ns
Block 8 write time: 25126221.632773 ns
Block 9 write time: 25086715.590922 ns
Block 10 write time: 25120815.797230 ns
Block 11 write time: 25071264.367816 ns
Block 12 write time: 25047217.801356 ns
Block 13 write time: 23867035.072207 ns
Block 14 write time: 25088696.728559 ns
Block 15 write time: 25087558.502800 ns
Block 16 write time: 25112159.151194 ns
Block 17 write time: 25032869.437076 ns
Block 18 write time: 25164425.582081 ns
Block 19 write time: 25079488.947834 ns
Block 20 write time: 23803767.167698 ns
Block 21 write time: 25056221.632773 ns
Block 22 write time: 25060913.056292 ns
Block 23 write time: 25162688.476275 ns
Block 24 write time: 25118358.974359 ns
Block 25 write time: 25091099.322134 ns
Block 26 write time: 25083797.229590 ns
Block 27 write time: 23706016.504568 ns
Block 28 write time: 25192990.863543 ns
Block 29 write time: 25078569.997053 ns
Block 30 write time: 25099977.600943 ns
Block 31 write time: 25086720.895962 ns
Block 32 write time: 25060146.183319 ns
```

Graphical representation of the output of DirectPointers() function



According to values we saw online we were expecting there to be 12 direct pointers. However that is not the case since after the first 12 block number the time it takes to write does not double as indicated by the cost for an indirect pointer

Conclusions:

We went through many iterations in the code that we implemented to gain a greater understanding of the file system for a Unix based machine. It seemed as though the logic for all of our experiments was sound but more often than not we did not get the results that we were expecting. We have a pretty firm understanding of the concepts of blocksize, prefetching, cache and direct pointers but it was difficult to measure them accurately. This most likely has to do with the many levels of abstractions that the OS has and maybe even optimizations that are taking place under the hood that are hard to account for.

The only time we had results coincide with the knowledge that we gained through research is when we used functions that did not depend upon timing mechanisms to file system calls or when we knew the explicit values for what we were trying to calculate ahead of time.

Estimated Time Spent on Project/Roles:

The project took a total of **45 hours** to complete. We individually did research by looking online and reading the Operating Systems book. This gave us an idea of how the file system is structured. After completing the initial research individually, we meet at the C4 lab and collaborated on potential solutions. We came to a general consensus and John started typing the code. We went through many iterations before coming to an algorithm that we were satisfied with. Finally with John's help I (Raj) typed up the report and created the images.

References:

ARK | Intel® Coreâ¢ i7-2600K Processor (8M Cache, up to 3.80 GHz). (n.d.). Retrieved November 27, 2014, from http://ark.intel.com/products/52214

Lseek. (n.d.). Retrieved November 27, 2014, from http://pubs.opengroup.org/onlinepubs/009695399/functions/lseek.html

What you Need to Know about Prefetching. (n.d.). Retrieved November 27, 2014, from https://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching

Fstatvfs. (n.d.). Retrieved November 27, 2014, from http://pubs.opengroup.org/onlinepubs/009695399/functions/fstatvfs.html

3.6. Prefetching. (n.d.). Retrieved November 27, 2014, from http://docs.roguewave.com/threadspotter/2012.1/linux/manual_html/ch_intro_prefetch.html