VOL. 2, NO. 9, September 2011 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

File Systems in Linux and FreeBSD: A Comparative Study

Kuo-pao Yang , Katie Wallace
Computer Science and Industrial Technology Department
Southeastern Louisiana University
Hammond, LA 70402
Corresponding Author: kyang@selu.edu

ABSTRACT

This paper compares file systems in Ubuntu Linux and FreeBSD and then analyzes the best utilization. The generic file systems, Extended File System (EXT2) of Linux and Fast File System (FFS) of FreeBSD operating systems, are evaluated using benchmark tests. It is proposed that a better file system could be assembled and implemented for improving performance and reducing bottleneck with consideration of modern intricacies.

Keywords: *Extended File System; Fast File System; Bonnie++; Iozone.*

1. INTRODUCTION

File systems of FreeBSD and Linux are very similar in general, but there are a few differences. This paper finds certain aspects of both file systems lacking in keeping up with performance for I/O devices. These file systems, long identified with FreeBSD and Linux are out of date, and cause bottleneck since they do not keep up with the performance levels of today's technology [7]. Therefore, it is proposed that a better file system could be implemented improving performance and reducing bottleneck.

CPU speeds and hard drive technology have been impressive growth in recent years [5]. In particular, data transfer rates of hard drives have improved dramatically with an increase in the performance of the data transfer paths and rotational speeds. However, processing capabilities have improved even more than disk access rates. At the same time, the demands for handling I/O and large data sets have also grown.

Both file systems have incorporated areas for updating their respective systems [4]. Even with these goals in place, the file systems are structured so that their performance leads to continuous I/O bottleneck. It has been shown that testing data can be altered to appear swayed. This benchmark test tries increase reliability in the test by reducing biased testing marks.

The rest of this paper is organized as follows. Section 2 discusses background information of both operating systems and compares the implementation of both file systems. Section 3 presents related work evaluation. Section 4 describes how we evaluated our system and presents the benchmark test results. Section 5 presents our conclusions and describes future work.

2. BACKGROUND

File management is one of the most noticeable components of operating systems [18]. The file system provides the mechanism for on-line storage and access to file contents, including data and programs. The operating system implements the abstract concept of a file by managing mass-storage media and the devices that control them [16]. It abstracts from the physical properties of its storage devices to define a logical storage unit. The operating systems maps files onto physical media and accesses these files via the storage devices. The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. There are different ways to structure file use to allocate disk space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage. Due to the various devices attached to a computer, the operating system uses a file system that provides a wide range of functionality to applications, to allow them to control all aspects of the devices. It should also provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key target is to optimize I/O for optimal compliance.

Since both FreeBSD and Linux have their origins in UNIX, their operating systems coexistence have similar file systems [14]. It handles multiple types of files by hiding the implementation details of single file types behind a software layer, the Virtual File System (VFS). This allows for a lack of duplication due to layering with object-oriented design principles. Both systems incorporate the Virtual File System, which makes all the files on the devices appear to exist in a single hierarchy.

The file system starts with a root directory with all other files down the tree-like structure. Every other file system is then mounted under the root file system [19]. Therefore, every directory appears to be part of the same disk. To gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear.

The Second Extended File System (EXT2) is the most widely used file system in the Linux community [4]. It provides standardized Unix file semantics and advanced features. The EXT2 supports standard Unix file types such as regular files, directories, device special files and symbolic links. EXT2 is able to manage file systems created on big partitions. While the original kernel code restricted the maximal file system size to 2 GB, recent work in the VFS layer have raised this limit to 4 TB. Thus, it is possible to use big disks without the need of creating many partitions. EXT2 provides long file names. It uses variable length directory entries. The maximal file name

VOL. 2, NO. 9, September 2011 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

size is 255 characters. This limit could be extended to 1012 if needed.

FreeBSD preferred native file system for local disks is the Berkeley Fast File System (FFS) [10]. It supports file systems up to 16TB and, depending on some parameters, files up to several terabytes in size. Files larger than 2GB on 32-bit platforms have been supported since BSD version 4.4 (1993) [11]. FreeBSD FFS and Linux EXT2 are generally very similar, which comes as no surprise considering that EXT2 was designed with FFS in mind.

File systems update their structural information called metadata by synchronous writes. Each metadata update may require many separate writes, and if the system crashes during the write sequence, metadata may be in inconsistent state [16]. At the next boot, the file system check utility called fsck must walk through the metadata structures, examining and repairing them. This operation takes a very long time on large filesystems. Moreover, the disk may not contain sufficient information to correct the structure. This results in misplaced or removed files.

3. COMPARISONS

The default Linux file system, EXT2, is influenced by the Berkeley Fast File System (FFS). However, EXT2 is notable since it is about the highest performance file system in general on any operating systems. In its general characteristics, EXT2 is very much like FreeBSD FFS with the "-o async" mount option. The Linux EXT2 file system gets its performance from having an asynchronous mount.

In terms of disk partitioning, the major difference between FreeBSD and Linux is that Linux uses the same partition scheme as IBM/Microsoft, while FreeBSD has its own partition scheme [12]. A place for this "slice" is sectioned on part of the disk. It is in this slice that the FreeBSD partitions are place. The terms "slice" and "partition" are used in opposite context for each operating systems.

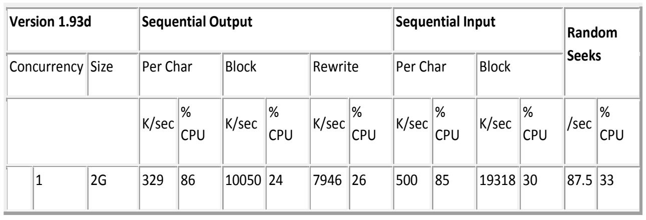
One difference between FFS and EXT2 that draws a lot of attention are the defaults for a particular safety or performance trade-off [13]. EXT2 defaults to writing metadata asynchronously, FFS to synchronously. Both file systems write normal data asynchronously. This gives EXT2 superior performance for operations involving metadata manipulations. Metadata refers to all data not contained within the files themselves: directory information, inodes, block allocation maps, etc. On the other hand, asynchronous handling of metadata exposes EXT2 to a substantial risk in case of a system crash or power outage. The file system may end up in an inconsistent state, which can entail severe data loss including loss of the complete file system. In comparison, FreeBSD uses soft updates. Ordered writes guarantee file system consistency and performance is similar to that of asynchronous metadata writing. The FreeBSD implementation was done by Kirk McKusick, the creator of FFS [1]. No structural changes to the on-disk filesystems are required.

4. BENCHMARK PERFORMANCE ANALYSIS

Benchmarks using Bonnie++ and Iozone have been performed for Ubuntu and FreeBSD. The versions were Ubuntu 8.10 (x86_64) with the Linux 2.6.27 kernel, X Server 1.5.2, GCC 4.3.2, GNOME 2.24, the EXT2 file-system, and FreeBSD 7.1 Beta 2 (AMD64) with X Server 1.5.2, GNOME 2.24, the FFS file-system, GCC 4.3.2, and Java 1.6.0_07-b02. Both operating systems were left in their default configuration. While testing for both operating systems comparison was completed on a single computer, with AMD hardware, the strongest performance is exhibited with Ubuntu 8.10.

Bonnie++ is a program to test file systems for performance. Bonnie++ tests file system operations for different applications used to unlike degrees. It gives a result of the amount of work done per second and the percentage of CPU time this took. For performance results, higher numbers are better. The step used here is to test creation, reading, and deleting many small files. One run like the following was issued: "bonnie++ -r 4096 -s 2048 -x1 -f | bon_csv2html > bonTest.html," to run the application. This says to start bonnie++ with a record size of 4k, this size is derived from the standard block size. The file size 2048 mb that is twice the size of the ram, it is to have one fast test performed for accuracy. The standard output is to become standard input and put through the bon_csv2 html application and that standard output is to be put into an .html file called bonTest. Note that the size of the test, record, number of threads and character all play a factor in determining the time length of the test. The larger is the longer the length of the test. One, rather inconclusive test, ran for three hours.

In the EXT2, Bonnie++ used a 2-Gigabyte file to perform the test shown in Figure 1. When writing the file by doing 2 billion *putc()* macro invocations, Bonnie recorded an output rate of 329 K per second. When writing the file by doing 2 billion *putc()* macro invocations, the operating system reported that this work consumed 86% of one CPU's time. This is not very good; it suggests either a slow CPU or an inefficient implementation of the *stdio* interface. When writing the 2-Gb file using efficient block writes, Bonnie recorded an output rate of 10,050 K per second. When writing the 2-Gb file using efficient block writes, the operating system reported that this work consumed 24% of one CPU's time. While running through the 2-Gb file just creating, changing each block, and rewriting, it, Bonnie recorded an ability to cover 7,946 K per second. While running through the 2-Gb file just creating, changing each block, and rewriting, it, the operating system reported that this work consumed 24% of one CPU's time. While reading the file using 2 billion *getc()* macro invocations, Bonnie recorded an input rate of 7946 K per second. While reading the file using 2 billion *getc()* macro invocations, the operating system reported that this work consumed 26% of one CPU's time. While reading the file using efficient block reads, Bonnie reported an input rate of 19,318 K per second. While reading the file using efficient block reads, the operating system reported that this work consumed 30% of one CPU's time. Bonnie created four child processes, and had them execute 4000 seeks to random locations in the file. On 10% of these seeks, they changed the block that they had read and re-wrote it. The effective seek rate was 87.5 seeks per second. During the seeking process, the operating system reported that this work consumed 33% of one CPU's time.

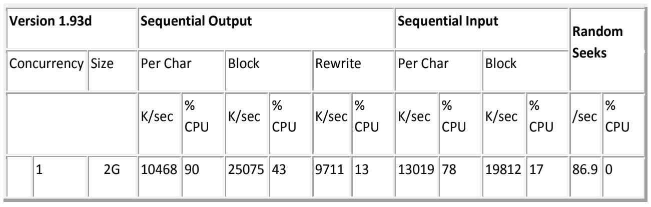


Version 1.93d		Sequential Output						Sequential Input				Random Seeks	
Concurrency	Size	Per Char		Block		Rewrite		Per Char		Block			
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU
1	2G	329	86	10050	24	7946	26	500	85	19318	30	87.5	33

Figure 1. Bonnie++ using Linux

For FFS, displayed in Figure 2, when writing the file by doing 2 billion *putc()* macro invocations, Bonnie recorded an output rate of 10,468 K per second. When writing the file by doing 2 billion *putc()* macro invocations, the operating system reported that this work consumed 90% of one CPU's time. When writing the 2-Gb file using efficient block writes, Bonnie recorded an output rate of 25,075 K per second. When writing the 2-Gb file using efficient block writes, the operating system reported that this work consumed 43% of one CPU's time. While running through the 2-Gb file just creating, changing each block, and rewriting, it, Bonnie recorded an ability to cover 9,711 K per second. While running through the 2-Gb file just creating, changing each block, and rewriting, it, the operating system reported that this work consumed 13% of one CPU's time. While reading the file using 2 billion *getc()* macro invocations, Bonnie recorded an input rate of 13,019 K per second. While reading the file using 2 billion *getc()* macro invocations, the operating system reported that this work consumed 78% of one CPU's time. While reading the file using efficient block reads, Bonnie reported an input rate of 19,812 K per second. While reading the file using efficient block reads, the operating system reported that this work consumed 17% of one CPU's time. Bonnie created four child processes, and had them execute 4000 seeks to random locations in the file. On 10% of these seeks, they changed the block that they had read and re-wrote it. The effective seek rate was 86.9 seeks per second. During the seeking process, the operating system reported that this work consumed 20% of one CPU's time.

The results of Bonnie++ are rather surprising, when looked at with all the previous benchmark results in mind. These results are typical of all those performed by Bonnie++. They show that FreeBSD lags behind Linux in performance. This margin that the EXT2 file system by which it won was larger than expected based on the differences between FreeBSD and Linux.

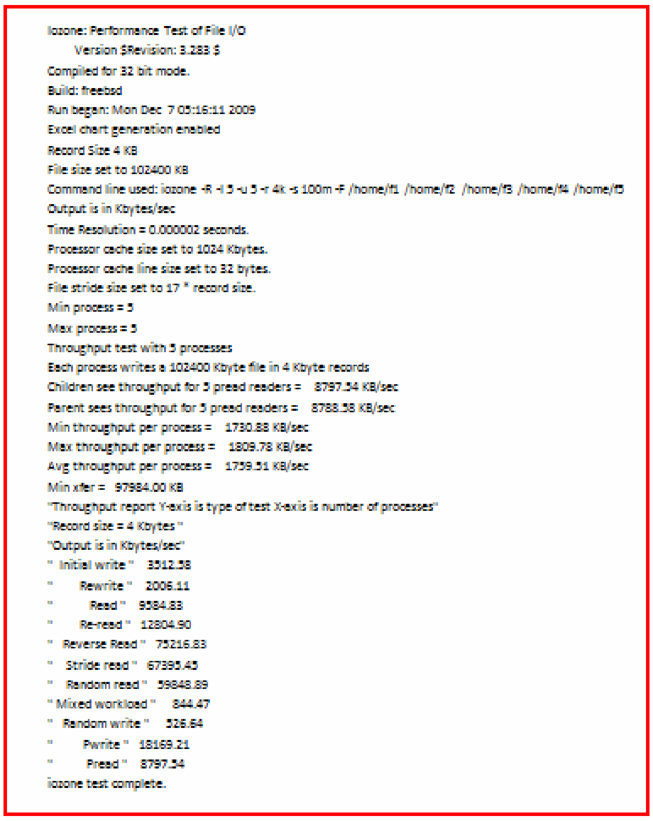


Version 1.93d		Sequential Output						Sequential Input				Random Seeks	
Concurrency	Size	Per Char		Block		Rewrite		Per Char		Block			
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU
1	2G	10468	90	25075	43	9711	13	13019	78	19812	17	86.9	0

Figure 2. Bonnie++ using Ubuntu

Iozone is a file system benchmark tool. The benchmark generates and measures a variety of file operations. Iozone has been ported to many machines and runs under many operating systems. Iozone seems to be able to perform a broader file system analysis than bonnie++. However, through this analysis, the comparisons were kept the came. The benchmark tests file I/O performance for the following operations: read, write, re-read, random write, random read. Iozone enables the administrator to receive a broad file system performance analysis to optimize his operating system for the best performance. A sample run is : "iozone -R -l 5 -u 5 -r 4k -s 100m -F /home/f1 /home/f2 /home/f3 /home/f4 /home/f5 | tee -a /tmp/iozone_results.txt &." This enables an excel component with an upper and lower bounds of 5 threads with a record size of 4k and a file size of 100mb. The capitalized F instructs iozone the temporary filenames used during testing and pipe the standard output to a .txt file called iozone_results.

Figure 3 below is a view of the standard output dumped on the terminal and text file after Iozone has executed. This particular execution had four threads total. The graphical comparisons of Write and Re-Read between the two systems are then as follows. Figure 4 shows that in the Write, the EXT2 begins with grater than 17,000 Kbytes per second and continues in that range. While the FFS, starts around 9,000 Kbytes per second and decreases. Figure 5 shows that while they start at the same range, 18,000 Kbytes per second, the FFS continually declines and the EXT2 varyingly increases.



```
Iozone: Performance Test of File I/O
        Version $Revision: 3.283 $
Compiled for 32 bit mode.
Build: freebsd
Run began: Mon Dec  7 05:16:11 2009
Excel chart generation enabled
Record Size 4 KB
File size set to 102400 KB
Command line used: iozone -R -l 5 -u 5 -r 4k -s 100m -F /home/f1 /home/f2  /home/f3 /home/f4 /home/f5
Output is in Kbytes/sec
Time Resolution = 0.000002 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
Min process = 5
Max process = 5
Throughput test with 5 processes
Each process writes a 102400 Kbyte file in 4 Kbyte records
Children see throughput for 5 pread readers =    8797.34 KB/sec
Parent sees throughput for 5 pread readers =    8788.38 KB/sec
Min throughput per process =    1730.88 KB/sec
Max throughput per process =    1809.78 KB/sec
Avg throughput per process =    1759.51 KB/sec
Min xfer =   97984.00 KB
"Throughput report Y-axis is type of test X-axis is number of processes"
"Record size = 4 Kbytes "
"Output is in Kbytes/sec"
"  Initial write "    3312.38
"       Rewrite "    2006.11
"          Read "    9584.83
"       Re-read "   12804.90
"   Reverse Read "   75216.83
"    Stride read "   67393.43
"    Random read "   59848.89
" Mixed workload "     844.47
"   Random write "     326.64
"        Pwrite "   18169.21
"         Pread "    8797.34
iozone test complete.
```

Figure 3. Standard Output of Iozone

VOL. 2, NO. 9, September 2011 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

5. CONCLUSIONS

In conclusion, both file systems are outdated and have already been revised for modern considerations. However, EXT2 appears to execute more efficiently than FFS in this limited amount of testing. Thus, Linux performs better than FreeBSD in these specific tests. FFS had a startling disappointment: it never scaled upward. In fact, every single test had better performance with fewer threads. Testing could have been swayed by the parameters set in the runnings. In addition, these tests do not by any means pretend to add additional substantive support to the optimization of any file system or operating system. Future applications include a greater variety of benchmark I/O performance bottleneck testing through new tests and techniques with RAID applications. Other options include applications that are much read and write intensive tests, measuring the performance of RAID levels, and the performance differences in changing the storage parameters. In the future, full, more inclusive benchmarks must be run for validity of the data.

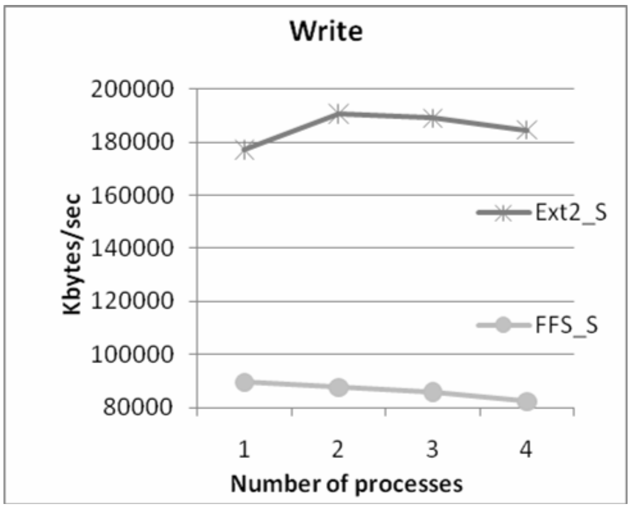


Figure 4. Iozone Write for EXT2 and FFS

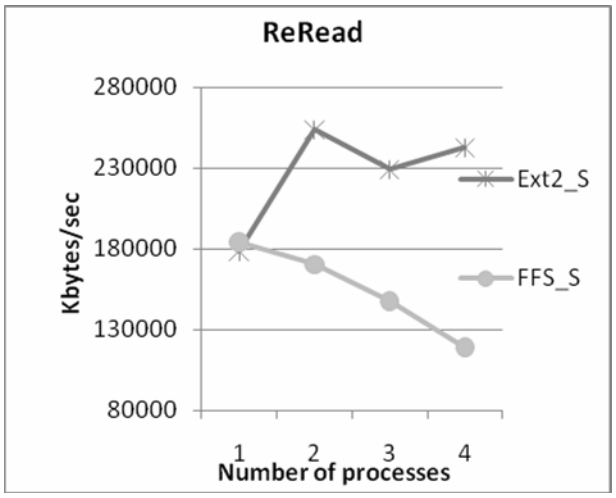


Figure 5. Iozone Reread for EXT2 and FFS

REFERENCES

[1] H. Allen, “UFS2 and Soft Updates make for a powerful combination,” Introduction to FreeBSD, PacNOG I Workshop, Additional Topics, Network Startup Resource Center, 2005, pp. 23.

[2] T. Bray and R. Coker, “bonnie++ - Linux man page,” Die.com, 2009.

[3] R. Card, T. Ts’o, and S. Tweedie, "Design and implementation of the second extended file system," Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9, 1994.

[4] B. Carrier, “File System Forensic Analysis,” Addison-Wesley, 2005.

[5] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian , A. Matsuoka, and L. Zhang, “Generalized file system dependencies,” Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007, pp. 307 - 320.

[6] M. Jambor, T. Hruby, J. Taus, K. Krchak, V. Holub, “Implementation of a Linux log-structured file system with a garbage collector,” ACM SIGOPS Operating Systems Review, v.41 n.1, 2007, pp. 24 - 32.

[7] R. McDougall and J. Mauro, ”The Sun Solaris UFS implementation chapter of the Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture,” Second Edition, ISBN 0-13-148209-2, 2007.

[8] M. K. McKusick, W. Joy, S. Leffler, and R. Fabry, “A Fast File System for UNIX,” Technical Report Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley,V.2, I. 3,1984, pp. 181 - 197.

[9] M. K. McKusick, K. Bostic, M. Karels, and J. Quarterman, "Local Filesystems," The Design and Implementation of the 4.4BSD Operating System, Addison-Wesley, 1996.

[10] M. K. McKusick, and G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference. V. 18, I. 2, 1999, pp. 1–18.

[11] M. K. McKusick, "Running "fsck" in the Background," Proceedings of the BSD Con 2002, May 2002, pp. 55–64.

[12] S. Pate, “UNIX Filesystems: Evolution, Design, and Implementation,” Wiley, 2002.

[13] M. Rosenblum, “The Design and Implementation of a Log-Structured File System,” The Springer International Series in Engineering and Computer Science, Springer, V.2 I. 1, 1992, 26 – 52.

[14] A. Silberschatz, P. Galvin, and G. Gagne, "Storage Management," Operating System Concepts , 7th edition, Wiley, 2004.

[15] A. Tanenbaum, "File Systems," Modern Operating Systems, 3rd edition, Prentice Hall, 2008.

[16] A. Tanenbaum, A. Woodhull, "File Systems," Operating Systems: Design and Implementation, 3rd edition, Prentice Hall, 2006.

[17] S. Tweedie, “Journalling the ext2fs Filesystem,” LinuxExpo ’98, 1998.

[18] E. Zadok, R. Iver, N. Joukov, G. Sivathanu, and C. Wright, “On incremental file system development,” ACM Transactions on Storage (TOS), v.2 n.2, May 2006, pp.161-196.