IOSTONE: A SYNTHETIC FILE SYSTEM PERFORMANCE BENCHMARK

Arvin Park

Richard J. Lipton

CS-TR-074-87

January 1987

IOStone: A Synthetic File System Performance Benchmark

Arvin Park
Richard J. Lipton

Department Computer Science
Princeton University
Princeton, New Jersey 08544

January 30, 1987

Abstract

We have developed a portable benchmark program which measures file system performance on *typical* system loads. Our program accomplishes this by generating a string of file system requests which is representative of measured system loads. Instead of isolating a particular aspect of file system performance such as disk access speed, or channel bandwidth, our program measures performance of the entire file system which includes components of disk performance, CPU performance on file system tasks, and buffer cache performance. This single metric can act as a valuable comprehensive measure of file system performance.

Measurements that we have made indicate that the balance between CPU performance and file system performance varies greatly across different computer systems. If an optimal balance between these two capabilities exists few system actually achieve this.

Keywords and Phrases - *Benchmarking, File Systems, Mass Storage, Performance Evaluation.*

1. Introduction

We desired to produce a single program which accurately simulates load conditions of *typical* file system usage. Of course it would be more accurate to assess each component of a file system's performance on each of a large number of different tasks. However, a single metric can provide a comprehensive measure of a system's performance under typical load conditions. This comprehensive measure can be used to rapidly compare the relative performance of different file systems.

Similar comprehensive metrics have been developed to measure system CPU performance. The *Whetstone* benchmark was developed to measure CPU performance on scientific applications [Curn76]. This program consisted of a collection of instructions which reflected the frequency of instruction usage measured in a large number of programs. The *Dhrystone* benchmark did the same for a collection of system programming codes [Weic84].

A number of programs exist which measure one or many components of file system performance. However, none thus far have attempted to simulate system performance under typical load conditions. Motivated by both the lack of a comprehensive file system benchmark, as well as the spirit of both the *Whetsone* and *Dhrystone* benchmarks, we have developed the *IOStone* benchmark which is a synthetic measure of file system performance under *typical* load conditions.

In Section 2 of this paper we describe the results of previous measurements of file system workloads. These help us establish a "*typical*" file system workload as well as an understanding for different benchmarking techniques. In Section 3 we describe the *IOStone* file system benchmark and how it correlates to different components of file system performance. Finally, in Section 4 we discuss limitations and applicability of our benchmark code. A version of our benchmark program written in the "C" programming language is presented in Appendix A.

2. Previous Studies and Benchmarks

Several types of file system studies have been undertaken. Many have dealt with automatic file migration [Lawr82] [Saty81] [Smit84]. A process where infrequently used files are automatically moved from rotating storage to cheaper forms of archival storage such as magnetic tape libraries. Although some of these studies are quite detailed, and they provide valuable insights into file size distributions and long term file referencing behavior, the type of measurements they perform are different from the ones we are interested in. They often measure static information on file lifetimes collected infrequently (daily) over long periods of time. Since our goal is to characterize a typical file system workloads. We are more concerned with dynamic information on file transfer and access requests.

A number of good studies have focused on file system performance for data base applications [Anon85] [Bora84] [Hawt79]. This is not surprising because database systems move large amounts of data to and from mass storage. Although these studies are quite detailed, and databases are a large subset of file system applications, again these studies are not quite suited to the workloads that we desire to measure.

Finally a number of studies have examined optimizations of file system performance [Hout85] [Hu86] [Krid83] [Maju86] [Oust85] [Park86] [Sher76] [Smith85]. Three of these studies ([Hu86]

[Oust85] [Smith85]) focused on dynamic measurements of file system workloads. The study by Smith measured file system loads for IBM machinery at Crocker Bank, Hughes Aircraft, and Stanford Linear Accelerator, while the study by Ousterhaut measured file system loads for the multi-user UNIX systems in at Berkeley. The study by Hu focused on a 68000 based single user workstation running the UNIX operating system.

Although these studies were performed across a range of installations and systems, several characteristics of the workload are the same for all of them: (1)Write operations account for about one third of all data transfer operations. (2) The majority of data transfer operations tend to be short. The UNIX system tends to magnify this fact because all accesses to the directory structure involve single block transfers. However, this tends to remain true even if accesses to the directory structure are disregarded. (3) Modest sized buffer caches reduce the number of data transfers between main memory and mass storage significantly [Oust85] [Smith85].

None of these studies accurately measured the contribution from paging activity, although some of Ousterhaut's measurements suggest that amount of this paging activity is rather small. Even though in present architectures the paging system contends for mass storage bandwidth with the file system, we are primarily interested in measuring file system performance, so we will restrict the scope of our evaluation to file system performance.

3. IOStone File System Benchmark

We desired to construct a simple, concise code that simulates a typical file system workload as closely as possible. Simplicity makes performance results easier to interpret as well as facilitating portability of the code across many different machines and operating systems. We view a file systems workload as a string of read and write requests. These read and write requests are of varying sizes and relative frequencies. As soon as the file system is done processing a requests it can begin processing the next one.

Of course, in reality the CPU may have to complete some processing before issuing another request to the file system. This causes the file system to remain idle for short periods of time. However, since we are not interested in assessing CPU performance on non-file system tasks, we assume that the file system never waits between operations.

Our benchmark code consists of a string of interspersed read and write requests. Two read operations occur for each write operation. This corresponds to the empirically verified write request frequency of 33.3 percent. The read and write requests vary in size. A histogram of the frequencies of different block sizes appears in Figure 1. This distribution was arrived at by approximating the distributions measured in the studies by Hu and Ousterhaut [Hu86] [Oust85].

Our program first creates a very large file, and then performs a series of read and write operations to random locations within the file. These random locations are calculated so that no operation will try to read past the end of the file. Creating and manipulating a large number of files from within a single program requires a great deal of overhead for some operating systems and is simply impossible for others. By accessing random locations within a single very large file we simulate a random series of accesses to a large number of smaller files.

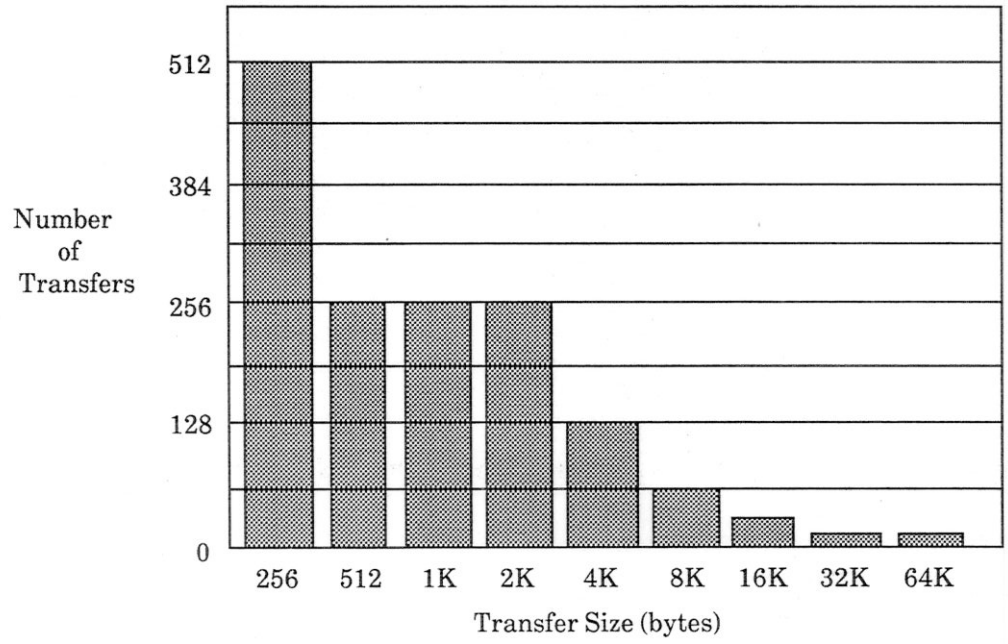

Figure 1: Histogram of IOStone Data Transfers

Our metric captures more of a file systems typical performance than simple maximum bandwidth or random access time measurements. This is because a typical file system's bandwidth varies non-linearly with the data transfer size.

Figure 2 presents results of file system performance measurements for read operations of different lengths that we conducted on a VAX - 11/750 Running the UNIX 4.3 BSD operating system. (This VAX-11/750 system has a DEC UDA50 disk controller connected to DEC RA80 disk drives. Details of this test are available in a previous paper [Park86].)

Maximum bandwidth measurements correspond to data points at the far right hand side of the graph where the file system bandwidth asymptotes towards its maximum value for the larger block sizes. Random access speed is represented by bandwidths on the left hand side of the graph. (The random access speed can be arrived at by examining bandwidth for a transfer request size that equals the block size of the file system. The random access speed is this block size divided by the transfer bandwidth of this block.) As can be seen from Figure 2, random access speed and maximum bandwidth do not tell the entire story. As the transfer size changes, so does the bandwidth. These intermediate values of bandwidth do not correlate well with either random access speed, maximum bandwidth, or a linear combination of these quantities. We capture the essence of this interplay between transfer request sizes and typical workloads with our benchmark which distributes its load across a representative distribution of transfer request sizes for both read and write operations.

We have run our the *IOStone* benchmark across a range of machines. These are ranked by IOStone performance in Table 1. Two of these machines are non-standard enough to mention here. The VAX - 11/785 that we tested was configured with an 8 megabyte buffer cache. This is about an order of magnitude larger than usual. The large buffer cache improved the system's *IOStone* performance considerably as can be seen from the figures in Table 1. One of the SUN 2's that we tested was

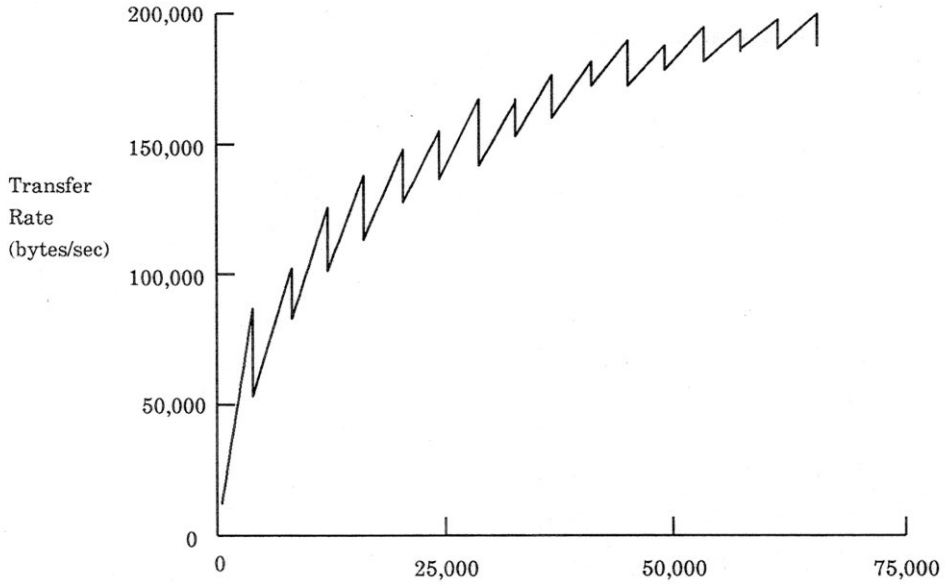

Figure 2: Test Results
Read Transfer Rate

Machine	Operating System	IOStones	Disk Controller	Disk Drive	Block/Frag Size (bytes)	Buffer Cache Size
VAX 11/785	UNIX 4.3 BSD	5650	DEC UDA50	DEC RA81	8K/1K	8M
VAX 8600	UNIX 4.3 BSD	2594	DEC UDA50	DEC RA81	8K/1K	1.6M
SUN 2 (68010)	UNIX 4.2 BSD	2000	SMD Multibus	Xilogics 450	4K/1K	304K
Pyramid 90X	OSX 2.5	1793	Pyramid 4051	Fujitsu 2351	8K/2K	408K
VAX 11/750	UNIX 4.3 BSD	1259	DEC UDA50	DEC RA80	4K/512	308K
IBM PC/AT	MS-DOS	704	IBM PC/AT	IBM PC/AT	512/512	20K
SUN 2 (diskless))	UNIX 4.2 BSD NFS	522	SMD Multibus	Xilogics 450	4K/1K	304K

Table 1: Machines Tested

diskless. All file I/O was channeled through a 10 megabit Ethernet to a remote file server. Note that

the diskless system still maintains about 26% of the performance of a comparable workstation with a disk.

Comparisons with random access speed appear in Figure 3. The systems we tested are plotted here with *IOStone* performance on the vertical axis and random accesses per second on the horizontal axis. Note that the machines tested all fall within a diagonal band on the graph. This means the ratio of *IOStone* performance to random access speed does not vary a great deal across different systems. This fact is not surprising because of the large number of small block transfers that were performed by the *IOStone* code. These small block transfers approximate random accesses.

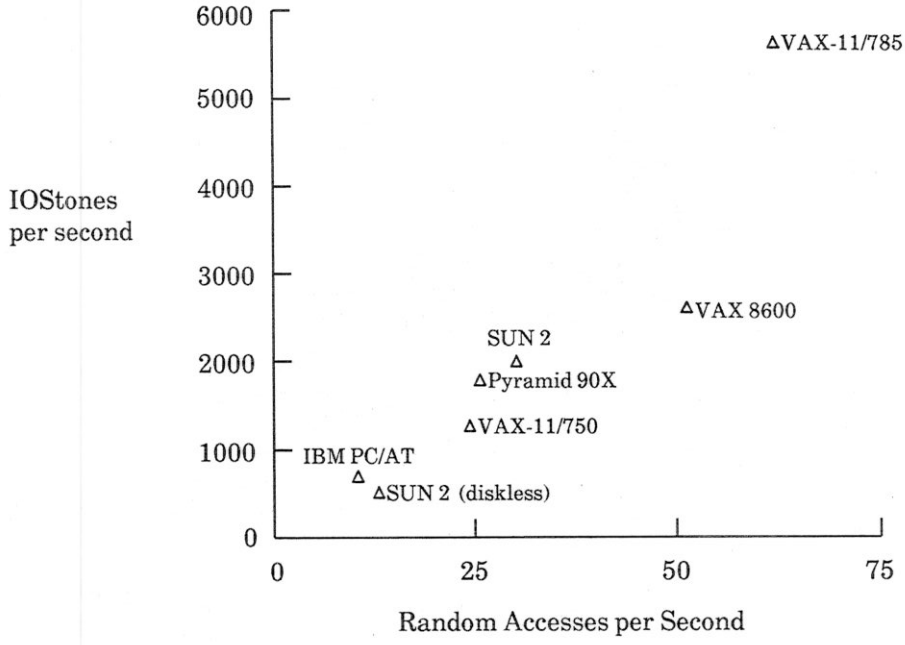

Figure 3: IOStone vs. Random Accesses per Second

A comparison with maximum bandwidth on read operations is presented in Figure 4. The systems are plotted here with *IOStone* performance on the horizontal axis and maximum read bandwidth on the vertical axis. Again the systems tend to lie within a diagonal band on the graph except for the VAX - 11/785 system. This 785 system has the large buffer cache which greatly improves random access performance but not maximum read bandwidth. The other systems seem to exhibit a fairly constant ratio of maximum read bandwidth to random access speed.

A comparison with *Dhrystone* measurements appears in Figure 5. The systems are plotted against *IOStone* performance on the horizontal axis and *Dhrystone* performance on the vertical axis. (Remember the *Dhrystone* code is a comprehensive measure of CPU performance on operating system tasks [Weic84]). There seems to be little consistency here. This indicates that the balance between CPU performance and file system performance varies greatly from system to system. If there does exist an optimal ratio of file system performance to CPU performance most of these systems are off the mark. The VAX 8600 seem particularly different from the other systems. It has an almost order of magnitude larger CPU performance, but only a slightly larger file system performance.

4. Limitations

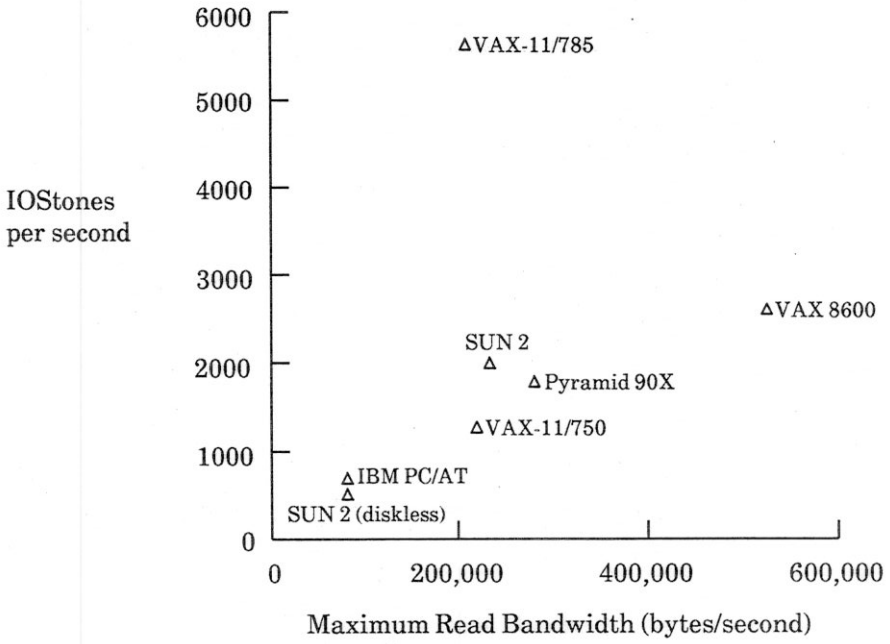

Figure 4: IOStone vs. Maximum Read Bandwidth

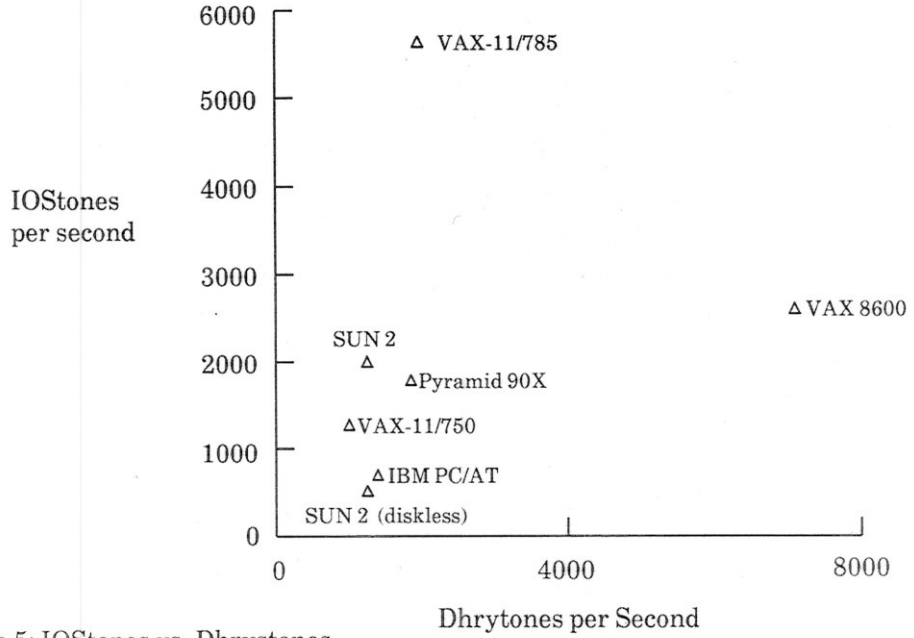

Figure 5: IOStones vs. Dhrystones

There are several limitations of our *IOStone* code. Our program measures file system performance on a typical system workload. The constitution of this typical workload was determined by a set of performance measurements from business, research, and development sites. Database and large scientific applications and can be expected to exhibit different workload characteristics, so performance on these tasks may not be accurately assessed on the basis of this benchmark.

We also do not attempt to measure system paging performance. Although paging and file systems contend for the same system resources, (disk drives, channels, etc.) we chose to concentrate exclusively on file system performance. In the interests of portability we have restricted ourselves to accessing a single very large file. One might argue that a system with an appropriately large buffer cache may make realize unrealistic performance gains from this restricted locality. However, Ousterhaut has noted [Oust85] that file system requests tend to exhibit a great amount of locality to begin with. Even a modest sized buffer cache greatly reduces the number of mass storage to main memory data transfers.

File creation and destruction operations are not performed in the benchmark. These do not occur during read operations which constitute the majority of transfer operations, and only a fraction of the write requests entail file creation or deletion operations anyway.

We only execute write operations to a existing file blocks. This differs from file extension write operations since new file blocks do not have to be allocated. However, the allocation process does not require great amounts of time anyway since the list of free blocks is frequently accessed and generally resides in the system buffer cache. Allocating a new block typically entails only a simple main memory lookup.

One might argue that the *IOStone* code does not generate a representative multiprogramming load since all data transfer requests are issued form a single process. One must recognize however that multiprogramming is a processor performance issue. The swapping overhead from processes that block on disk requests does not interest us since we consider this this is a CPU activity which is unrelated to file system functioning. We would like our benchmark to apply to both multi-user systems and single user workstations.

Conclusions

We have developed a benchmark program which simulates a workload that closely corresponds to measurements of real system workloads. Our code can act as a valuable tool to evaluate file system performance.

References

[Anon85] Anon et. al., "A Measure of Transaction Processing Power", *Datamation*, April 1, 1985, pp.112-118.

[Bora84] H. Boral, D. J. Dewitt, "A Methodology for Database Performance Evaluation", *Proceedings of the 1984 ACM - SIGMOD International Conference on Management of Data*, May 1984, pp. 176-185.

[Curn76] H. J. Curnow, B. A. Wichman, "A Synthetic Benchmark", *Computer Journal*, Volume 19, Number 1, February 1976, pp. 43-49.

[Hawt79] P. Hawthorn, M. Stonebraker, "Performance Analysis of a Relational Database Machine", *Proceedings of the 1979 ACM - SIGMOD Conference on Management of Data*, pp.1-12.

[Hout85] G. E. Houtekamer, "The Local Disk Controller", *Proceedings of the 1985 ACM - SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Volume 13, Number 2, August 26-29, 1985, pp. 173-182.

[Hu86] I. Hu, "Measuring File Access Patterns in UNIX", *Proceedings of the 1986 ACM - SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Volume 14, Number 2, May 27-3, 1986, pp. 15-20.

[Krid83] B. Kridle, M. K. McKusick, "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Technical Report.

[Lawr82] D. H. Lawrie, J. M. Randal, R. R. Barton, "Experiments with Automatic File Migration", *Computer*, Volume 25, Number 7, July, 1982, pp. 45-55.

[Maju86] S. Majumdar, R. B. Bunt, "Measurement and Analysis of Locality in File Referencing Behavior", *Proceedings of the 1986 ACM - SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Volume 14, Number 1, May 27-3, 1986, pp. 180-192.

[Oust85] J. K. Ousterhaut, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, J. G. Thompson, "A Trace Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the Tenth Symposium on Operating Systems Principles*, 1985, pp. 15-24.

[Park86] A. Park, R. J. Lipton, "Models and Measurements of File System Performance", Technical Report Number 086-067, Princeton University, Department of Computer Science, Princeton, N. J., 08544.

[Saty81] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes", *Proceedings of the Eighth Symposium on Operating Systems Principles*, 1981, pp. 96-108.

[Smit84] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms" , *IEEE Transactions on Software Engineering*, Volume SE-7, Number 4, July 1981, pp. 403-417.

[Smit85] A. J. Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems*, Volume 3, Number 3, August 1985, pp. 161-203.

[Weic84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", *Communications of the ACM*, Volume 27, Number 10, October 1984, pp. 1013-1030.

Appendix A: IOStone Code (C language version)

```
/*
*    "I/O Stone" Benchmark Program
*
*    Written by:    Arvin Park (park@princeton)
*                   Department of Computer Science
*                   Princeton University
*                   Princeton, New Jersey 08544
*                   (609) 452-6304
*
*    Version:       C/1
*    Date:          12/10/86
*
*    Defines:If your version of "C" does not include a time(2)
*            function, define NOTIME. Use a stopwatch to measure
*            elapsed wall time. Divide 400000 by the elapsed time
*            to get the correct number of iostones/second.
*
*    To compile:    cc -O io.c -o io
*
*    Note:  [1] This program should be run without other processes
*           competing for system resources. Run it in the dead of
*           night if you have to.
*
*           [2] This program uses 4 megabytes of disk space. Make
*           sure that at least this much space is available on
*           your file system before you run the progam.
*
*    Results:If you get results from a new (machine/operating
*            system/disk controller and drive) combination please
*            send them to park@princeton. Please include complete
*            information on the machine type, operating system,
*            version, disk controller, and disk drives. Also make
*            a note of any system modifications that have been
*            performed.
*/

#define FILESIZE (4L*1024L*1024L)        /*size of file in bytes*/
#define MAXBUFFERSIZE (64L*1024L)        /*maximum buffer size*/
#define NBLOCKSIZES 9                    /*number of different block sizes*/
#define SEED 34710373L                   /*random number generator seed*/
#define CONST 100000L                    /*iostone normalization constant*/
#define ITER 4                           /*number of iterations of the code*/
```

```
/*Define only one of the following two.*/
/*#define NOTIME                          /*define if no time function in library*/
#define TIME                              /*Use time(2) function*/

char buffer[MAXBUFFERSIZE];               /* a temporary data buffer*/
char *filename = "/tmp/iostone__temp__file"; /*name of temporary file*/
unsigned int nbytes;                      /*number of bytes transfered*/
int fd;                                   /*file descriptor*/
long offset;                              /*file offset*/
int i,j,k;                                /*counter variables*/
long bsize[NBLOCKSIZES];                  /*array for different block sizes*/
int bfreq[NBLOCKSIZES];                   /*number of accesses for each block*/

#ifdef TIME
    long        time();
    long        starttime;
    long        totaltime;
#endif

main() {
    init();

                                          /*start timing*/
#ifdef NOTIME
    printf("start timing\n");
#endif
#ifdef TIME
    starttime = time(0);
#endif

    for(k=0;k<ITER;k++)                   /*perform string of file operations*/
        readswrites();

                                          /*stop timer*/
#ifdef NOTIME
    printf("stop timing\n");
#endif
#ifdef TIME
    totaltime = time(0) - starttime;
    printf("total time = %ld\n", totaltime);
    if(totaltime!=0)
        printf("This machine benchmarks at %ld iostones/second\n",
        (long) (CONST*ITER)/totaltime);
#endif
```

```
}

init() {
                    /*create a temporary file*/
    if((fd = creat(filename,0640)) < 0) {
        printf("init: Cannot create temporary file\n");
        exit(1);
    }
                    /*To both read and write the file*/
                    /* it must be closed then opened*/
    close(fd);
    if((fd = open(filename,2)) < 0) {
        printf("init: Cannot open temporary file\n");
        exit(1);
    }
                    /*Unlink the file so that it will*/
                    /* disappear when the program*/
                    /*terminates.*/
    unlink(filename);

    lseek(fd,0L,0);          /*write initial portion of file*/
    for(i=0;i<(FILESIZE)/4096;i++){
        if((nbytes = write(fd,buffer,4096)) < 0) {
            printf("init:error writing block\n");
            exit(1);
        }
    }

                    /*set file block sizes and access*/
                    /*frequencies.*/
    bsize[0] = 256; bfreq[0] = 128;
    bsize[1] = 512; bfreq[1] = 64;
    bsize[2] = 1024; bfreq[2] = 64;
    bsize[3] = 2048; bfreq[3] = 64;
    bsize[4] = 4096; bfreq[4] = 32;
    bsize[5] = 8192; bfreq[5] = 16;
    bsize[6] = 16384; bfreq[6] = 8;
    bsize[7] = 32768; bfreq[7] = 4;
    bsize[8] = 65536; bfreq[8] = 4;

    random(SEED);            /*initialize random number generator*/
}
```

```
readswrites() {
    for(j=0;j<NBLOCKSIZES;j++) {
        for(i=0;i<bfreq[j];i++){
            offset=(long) ((random()%(FILESIZE/bsize[j]))*bsize[j]);
            lseek(fd,offset,0);
            if((nbytes = read(fd,buffer,bsize[j])) < 0) {
                printf("readswrites: read error\n");
                        exit(1);
            }
            offset=(long) ((random()%(FILESIZE/bsize[j]))*bsize[j]);
            lseek(fd,offset,0);
            if((nbytes = read(fd,buffer,bsize[j])) < 0) {
                printf("readswrites: read error\n");
                exit(1);
            }
            offset=(long) ((random()%(FILESIZE/bsize[j]))*bsize[j]);
            lseek(fd,offset,0);
            if((nbytes = write(fd,buffer,bsize[j])) < 0) {
                printf("readswrites: write error\n");
                exit(1);
            }
        }
    }
}
```