

An Implementation of Large Files for BSD UNIX

Dave Shaver, Eric Schnoebelen, George Bier – CONVEX Computer Corporation

Abstract

The design of the ConvexOS¹ filesystem, based on the BSD Fast File System, allows for a theoretical maximum file size of about 4402G² with a 4K filesystem block size (or about 64T with 8K blocks.) Unfortunately, the actual limit of the CONVEX filesystem has been 2G-1 because key kernel values and file offset pointers are 32-bits in size. This is a problem shared by many other UNIX³ vendors. This paper describes the path CONVEX has taken to implement files and filesystems larger than 2G. The implementation is based on a new set of 64-bit system calls and new library interfaces; it requires no changes to the on-disk i-node representation. The large file programming models and the kernel and utilities changes are described. Measurements of read and write I/O rates are presented and show that there is little performance penalty for manipulating large files using the chosen implementation.

Introduction

The Berkeley Software Distribution (BSD) Fast File System (FFS) is a relatively high-performance filesystem for UNIX. (See [McKusick84] and [Leffler89] for a full description of the FFS.) The key data structure of each file is the i-node. Data within the FFS is referenced via the i-node and is stored in both direct blocks and in single, double, and triple indirect blocks. Within the i-node there are twelve direct block pointers, each of which point to a block of on-disk file data. The single indirect pointer is a reference to a filesystem block full of direct pointers. Double and triple indirect pointers are implemented in a similar manner. In theory, by using triple indirect blocks, the FFS supports very large files (up to 64T with 8K filesystem blocks, for example.) The on-disk FFS i-node also has 64 bits set aside to hold the file size. However, most FFS implementations do not use the triple indirect blocks [Leffler89] and only use 32 of the 64 bits in the on-disk i-node to hold the file size.

Although CONVEX's on-disk i-node structure supports very large files, our system call interface limits file sizes to 2G-1. The key variable that causes this limitation is the 32-bit file offset pointer. The file offset pointer must be able to point to any location within a file; thus a signed, 32-bit file offset pointer implicitly limits file sizes to 2G-1. At CONVEX, a leading supplier of air-cooled supercomputers, we have addressed the issue that all high performance vendors will eventually face: creating files larger than 2G-1. This issue is similar to the transition from a 16-bit, block-oriented interface under UNIX V6 (`seek`), to a 32-bit interface under UNIX V7 (`lseek`).

In the next section we describe the programming models we considered to overcome the 32-bit file offset problem. The advantages and disadvantages of each scheme are discussed and the chosen implementation is described in detail. Although the selected implementation does not have the most elegant C programming interface, it does meet our primary goal that FORTRAN programs only need to relink with the latest version of the FORTRAN libraries to manipulate large files. Subsequent sections describe the changes we made to the kernel and the utilities to support large files. Finally, performance evaluation results are given, showing that the I/O rates for large, sequential reads and writes degrade only slightly as file sizes grow beyond 2G.

Programming model alternatives

The problem we needed to solve was the implications of changing the file offset pointer from 32 to 64 bits at the system call level. The major constraints on our solution were:

¹ConvexOS and CONVEX are trademarks of CONVEX Computer Corporation.

²Throughout this paper we use K for kilobytes, M for megabytes, G for gigabytes, and T for terabytes.

³UNIX is a registered trademark of UNIX Systems Laboratories.

1. Not changing the on-disk i-node, and
2. Remaining backwards compatible with binaries and sources that do not require large files

Had we not been able to meet the first constraint, we probably would not have implemented large files under our current FFS-based filesystem. This constraint arose out of concern for the impacts our implementation would have on our customers.

The second constraint on our solution implied two things. First, we needed to solve the burdensome issue of how the kernel treats applications that do not comprehend large files, yet attempt to manipulate an existing large file. Second, we had to provide both the existing 32-bit versions of the system calls as well as new 64-bit versions.

We coined the term “large file unaware” to describe an application that does *not* know files may be larger than 2G. Because of the first major constraint on our solution, we wanted a large file unaware application to view an existing large file as a file 2G-1 in size. For example, if an unaware application opens an otherwise quiescent large file in `O_APPEND` mode, it receives an `EINVAL` error at `write()` time. This normal appearance of large files to unaware applications extends beyond just the `read()` and `write()` semantics at the 2G-1 boundary. We do not want an unaware application to `write()` to or `read()` from a location to which it can not `lseek()`. This arises out of a concern for maintaining our POSIX compliance for unaware applications. Once we had chosen how an unaware application would be treated if it attempted to manipulate a large file, we addressed the programming model alternatives for both FORTRAN and C.

Since FORTRAN I/O is record based and accessed only through library routines, we were able to modify the libraries to exclusively use the new 64-bit system calls. Thus FORTRAN programs become large file aware automatically, simply by relinking. However, there is still an implicit restriction on file sizes under FORTRAN since record numbers must fit in a 32-bit data type. Although a FORTRAN application can still not create files that contain more than 2^{32} records, they can build files up to 1T-512 as long as they do not exceed the record limit.

Unlike the seamless introduction of large files to FORTRAN, it was difficult to choose the C large file programming model. The existing standards POSIX.1 and ANSI C were consulted to see if they might influence our large file interface, since ConvexOS is POSIX.1 compliant in the ANSI C environment (see [CONVEX89-1][CONVEX89-2] for more information.) The ANSI C standard [ANSI90] requires a `long` as the data type passed to `fseek()` and the type returned by `ftell()`. However, it places no restrictions on the data type used and returned by `fsetpos()` and `fgetpos()`. The POSIX.1 standard [IEEE88], when used in the ANSI C environment, requires that `off_t` be an integral, arithmetic, type. With these restrictions in mind, several programming models were considered, including the following:

1. Using segments or block offsets in addition to straight byte offsets during a `lseek()` call. This is how UNIX V6 solved the problem of specifying an offset larger than allowed by the PDP-11's 16 bit integers.
2. Add mode bits on the `a.out` image that allow the kernel to determine if the application is large file aware.
3. Create new system calls that use a larger file offset, but make them invisible to the programmer.
4. Create new system calls that use a larger file offset, but make them visible to the programmer.

We felt that the first model, using segments, is too dissimilar to modern UNIX-like operating systems to be seamlessly integrated into ConvexOS. We also noted that the UNIX V6 segmented solution with `seek()` was later dropped in favor of `lseek()`.⁴ Thus, in an attempt to learn from UNIX history, we didn't want to create another block-oriented interface.

The second model involves a new mode bit within the application. Although ConvexOS has used the mode bit solution in the past—most notably for our POSIX.1 support—we felt that this model required too much effort to be implemented in the time available. Specifically, implementing this solution requires kernel, compiler, loader, and library work.

With the segmented interface and mode bit models eliminated, only two alternatives for the C large file programming model were seriously considered. The first alternative was to hide the 64-bit versus 32-bit

⁴[Kernighan], p. 164-5.

system call issue from the user, as we did in the FORTRAN programming model. The second alternative, and the one implemented, is to expose the new 64-bit system calls to the user.

The first alternative, hiding the 64-bit versus 32-bit system calls, has the advantage that existing ANSI C-conforming code can be recompiled and instantly becomes large file aware. However, this requires that our `long` type become a 64-bit data type. A disadvantage of this model is that non-ANSI C-compliant code that assumes `sizeof(int) == sizeof(long)` might not execute correctly when compiled in ANSI C mode.

The second alternative forces the programmer to explicitly call the new 64-bit system calls for large file access. This has the advantage that, without compiler changes, we could implement, test, and deliver a large file product in the time that was available. The data type for the 64-bit file offset pointer that is used by the new system calls is an `off64_t`. This is a `typedef` to our compiler's preexisting 64-bit data type, the `long long`. The disadvantage is that this large file C programming model is not compliant with either ANSI C or POSIX.1. However our chosen C programming model meets most customer needs.

From a C programmer's standpoint, both proposed C programming models require about the same amount of work to make a program large file aware. The first requires finding potential standards violations in the source, while the second requires recoding the application to use the new system call interface. Thus, potentially, both models require code modifications.

A key point we kept in mind while developing our programming models for C and FORTRAN was that having large *filesystems* benefits even large file unaware applications. Although an unaware application can not manipulate large files, large filesystems can ease administration of disk resources. In our environment, customer demand for large filesystems is certainly stronger than demand for large files.

Kernel changes

Since the chosen programming model increases the file offset pointer to 64 bits, the new maximum file size is theoretically $2^{63} - 1$ bytes. However, because the design required that the on-disk filesystem i-node representation not change, we were unable to achieve this maximum size. The internals of the CONVEX kernel and filesystem may perform I/O in blocks as small as 512 bytes; thus a file is logically broken into sequentially-numbered 512 byte blocks. Since these block numbers are stored as a signed 32-bit value, the largest file that can be created is limited to the maximum number of 512 byte blocks this value can represent. Therefore, the maximum large file size becomes $(2^{31} - 1) \times 512 = 2^{40} - 512 = 1\text{T} - 512$. This limitation is considered acceptable given current disk technology and customer applications. We believe that by the time the 1T limit becomes inhibitive, we will either have a new filesystem or an entirely new OS technology. Also note that increasing the size of the block pointer to 64 bits requires work within the filesystem itself and within other kernel subsystems. This task could easily quadruple the work necessary to implement large files.

Since no major filesystem modifications were desired, we did not need to take actions as radical as MSS-II did to implement large filesystems under Amdahl's UTS [NASA90]. The UTS filesystem is based on the standard System V filesystem, thus the MSS-II work needed to change the on-disk filesystem structure. Also, since ConvexOS already supported disk striping, we did not need to solve the problem of building filesystems larger than a single physical disk partition (see [CONVEX91] and [Landherr91] for a full description of our disk striping implementation.) Given the limitations of our proposed large file implementation, the kernel work broke down into two main tasks:

1. Programming interface changes needed to implement our programming model, and
2. Internal filesystem changes.

Each of these tasks is described below in detail. Note that both performance tuning of the filesystem and the scalability of existing filesystem algorithms were not addressed during this project.

The programming interface was enhanced to include new 64-bit versions of key system calls. The new system calls are: `lseek64()`, `truncate64()`, `ftruncate64()`, `stat64()`, `fstat64()`, and `lstat64()`. Each call matches the functionality of its 32-bit counterpart, but works with files larger than 2G. We added new `open()` (`O_LARGEFILE`) and `fcntl()` (`FLARGEFILE`) large file flags. If an application uses either flag during the appropriate system call, it will have access beyond a file's 2G point; without the flag, the application is considered large file unaware and it will view an existing large file as a 2G-1 file. Applications accessing large

files via NFS are considered large file unaware, thus only the first 2G-1 bytes of a large files are accessible via NFS.⁵ When `lseek64()` is called, it sets the `FLARGEFILE` bit as a side effect since any application that uses this system call is considered implicitly large file aware.

The internal filesystem changes involved adding 64-bit versions of the key system calls and correctly interpreting the file size field of the existing on-disk i-node. To eliminate duplication of code, both `lseek()` and `lseek64()` are stubs that call `seek_internal()`. `seek_internal()` knows the type of seek desired, works exclusively with 64-bit offsets, and enforces the old 2G-1 file limit on large file unaware applications. In a similar fashion, `truncate()` and `truncate64()` call `truncate_i()`, while `ftruncate()` and `ftruncate64()` call `ftruncate_i()`.

Correctly interpreting the file size field of the i-node is important. The on-disk i-node already used a `quad`⁶ for the file size, thus we did not need to enlarge it. Before the implementation of large files, one of the two `longs` in the `quad` was unused since only 32 bits were needed. Unfortunately, due to an error originally made while porting the BSD kernel to the `CONVEX C` series architecture, the wrong `long` was in use. Thus, a macro was written to swap the two `longs` in the `quad` each time the value is read from or written to disk. However, after the `quad` has been read from disk and the `longs` have been swapped, the value is strictly treated as an `off64_t`. This movement from 32 to 64 bit offsets within the kernel is not a performance issue since the `CONVEX C`-series architecture has 64-bit scalar registers. An additional feature is the ability to disallow the creation of large files on a per-filesystem basis. Besides the new functionality, we spent time changing or adding type casts within the filesystem code.

Testing of the new kernel and library functionality took about six programmer weeks and was accomplished using normal filesystem semantics regarding holes in files. Since on-disk data blocks are not allocated for a hole, we could create very large files on a small physical disk stripe. Although much of our development was carried out with ten 1G drives striped together in various ways, a single filesystem of over 80G of physical storage was created since large file support was released.

Library work to support large files broke down into these tasks:

- Adding `fseek64()` and `ftell64()` to `stdio`.
- Enhancing `fgetpos()` and `fsetpos()`.
- Adding support for the new `l` flag to `fopen()`, `fdopen()`, and `freopen()`.
- Adding the remaining entry points for the new system calls.

The total effort required for the kernel work was about 18 programmer weeks. C library work took about three programmer weeks while FORTRAN support took four programmer weeks.

Utility changes

The major problem in making programs large file aware is caused by code that assumes the key system calls and library functions accept and return a `long`. To resolve this assumption requires careful review of the code. The number of utilities that we made large file aware for the first release was limited because of time constraints. We found the following basic set of utilities adequate:

Utilities for creating and maintaining filesystems: `fsck`, `ncheck`, `mkfs`, `fsirand`, `mount`, `newfs`, `dumpfs`, `newst`, `putst`, `getst`,⁷ `dump`, `xdump`,⁸ and `restore`.

Utilities identified as essential for the reasonable use of large files: `ls`, `cp`, `mv`, `rcp`, `ftp`, `find`, `tail`, `cat`, `dd`, `chkpnt`, `restart`,⁹ `compact`, `cmp`, `tar`, and `cpio`.

⁵This was due to concern for compatibility with other NFS implementations and 32-bit pointers within the NFS definition.

⁶A `quad` is two `longs` wide, for a total of 64 bits.

⁷The `*st` utilities are used for maintaining our implementation of disk stripes.

⁸`xdump` is a fast dump utility fully described in [Polk88].

⁹Both `chkpnt` and `restart` are used in the ConvexOS implementation of a checkpoint/restart system.

Since we were adding support for large filesystems in addition to support for large files, it was necessary to enhance the utilities used for creating and maintaining filesystems. Thus we considered it essential, from project inception, that all existing filesystem manipulation and maintenance utilities be large file aware.

It was difficult to choose the subset of remaining utilities to make large file aware. Given our project schedule and staffing, it was necessary that only a minimal set of utilities be made fully large file aware. There was neither time nor resources to examine and enhance all 400 utilities that are part of ConvexOS.

`ls` is an obvious choice to be made large file aware since it displays the sizes of files. `cp` and `mv` are two other obvious candidates for large file awareness, although both were further enhanced to preserve sparse files (“holes”) during copying. Both `rnp` and `ftp` were enhanced and are capable of transferring large files over the network, although neither preserves sparse files. `find` was also made large file aware since it includes both a `-ls` and `-size` option.

`tail` was made large file aware because it allows users to look at the end of a file, and it allows the continual monitoring of a file using the `-f` option. The corresponding `head` utility was left large file unaware since it only works with the first 2G of a file.

Since we considered making our shells (`sh`, `csh`, and `ksh`) large file aware an immense task, I/O redirection with large files is not implemented. This issue can be avoided since, with a data source and a data sink, a pipeline—that has no size limits—can be created. For example, the output of an application can be piped into a large file aware data sink, that can then write the output to a large file. We provide both data sources (`cat` or `dd`) and a data sink (`dd`) that are large file aware. Thus, rather than using shell redirection like this:

```
% application > large_file
```

the user pipes application output into a data sink like this:

```
% application | dd of=large_file
```

Both `chkpnt` and `restart` required changes since, in conjunction with the kernel, they recover and restore file offsets to/from in-core process images. If these utilities had been left large file unaware, it would be impossible to checkpoint applications that use large files.

`compact` and `cmp` were added late in the project. Although neither one is efficient enough to be pleasantly used with large files, the effort required to make them large file aware shows the relative ease of making an application large file aware under our chosen programming model.

`tar` and `cpio` were special enhancements. We wanted them to be large file aware since they give users some form of file level archiving other than the `dump` and `restore` suite. Because of limitations in the archive header formats imposed by [IEEE88], these two utilities are limited to archiving files smaller than 8G.

There are some seemingly useful tools, such as editors and other data manipulation tools, noticeably missing from the list of large file aware utilities. We feel that interactive editors are not useful on files larger than 1G, even if the user does have enough space in `/tmp`. Our reasoning is that large files are manipulated by applications, not by manual editing. Also, in general, the editing of such large files should be batch-based, using either specially written tools, or `sed`.

Further, tools such as `sed` and `grep` were not enhanced since they are designed to read from standard input and write to standard output. When used in this mode they are inherently large file aware. It is simple enough to create a pipeline with a large file aware data source at one end, the data manipulation tool(s) in the middle, and a large file aware data sink at the other end.

Finally, utilities that examine filesystem data seem to be prime candidates to become large file aware, but are frequently found to not require modification when inspected more closely. For example, many utilities such as `du` and `df` do their computations in terms of filesystem *blocks*, not *bytes*. As we continue to use our large file implementation, we will find and fix additional utilities that need to be large file aware.

The process of making a program large file aware generally takes the following steps:

- Cleaning up the source to compile in the ANSI C-conforming mode of our compiler. This is necessary since large files are only available when using the ANSI C mode of our compiler. They are only available in this mode since, generally, we do not add new functionality to the backwards compatible mode of our compiler.

- Correcting incorrect data type assumptions. For example, this includes converting code to use `off_t` for file offsets (for ANSI C conformance), correcting the assumption that the number of bytes in a filesystem can be expressed as a 32-bit value, etc.
- Changing instances of `creat()` into calls to `open()` with the `O_CREAT` flag.
- Adding either the new `O_LARGFILE` flag on calls to `open()` or the new `l` modifier on calls to `fopen()`, `fdopen()` or `freopen()`. Both additions give the resulting file descriptor access to large files.
- Changing calls to `fstat()` or `lstat()` into calls to `fstat64()` `lstat64()`, with the corresponding change of the `struct stat` to a `stat64_t`.¹⁰ Any use of the `st_size` field must be verified to make sure proper data types are being used.
- Changing calls to `fseek()` or `lseek()` into calls to `fseek64()` or `lseek64()`, with the corresponding change of the offset parameter from an `off_t` to an `off64_t`.
- Changing calls to `truncate()` or `ftruncate()` into calls to `truncate64()` or `ftruncate64()`. This also requires that the offset parameter passed to these functions be an `off64_t`, either directly or via a cast.

The total effort required for utilities conversion was about 20 programmer weeks. The most difficult utilities to convert were `fsck`, `dump`, and `xdump` since they had intimate knowledge of the filesystem. Testing of the utilities took about nine programmer weeks.

Performance evaluation

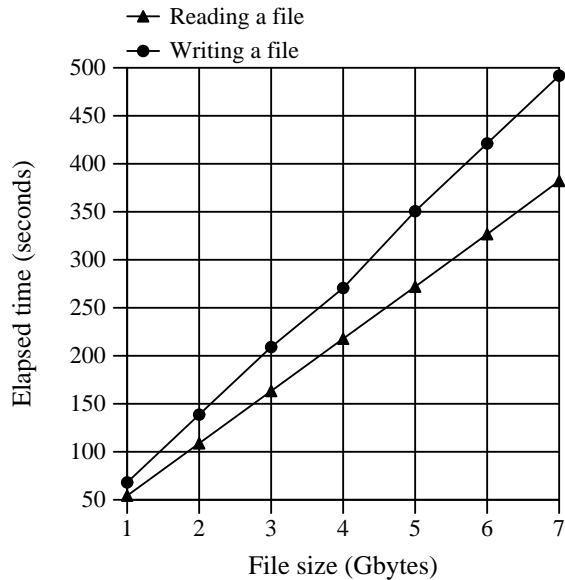
At the time large files were implemented, CONVEX's high performance disk drive was the Seagate Sabre 5 HP2, a 1.1G IPI-2 drive. After formatting, a drive has about 800M of data space available. Thus, to create a filesystem larger than 2G, requires striping multiple drives together. On large sequential reads and writes, each drive in a stripe will achieve a throughput of 5.2M/sec until the maximum number of drives that the CPU can drive at full speed is reached.

I/O performance is evaluated using a C program that does large sequential reads or writes. Parameters for the program are the size per read or write system call and the total file size to read or write. The total number of system call requests per run is obtained by dividing the file size by the size per request. The read times reported avoid interference from the buffer cache by flushing the cache before each measurement run. Writes go through the buffer cache, but include the time to do a `fsync()` system call, guaranteeing that the data has been transferred to disk.

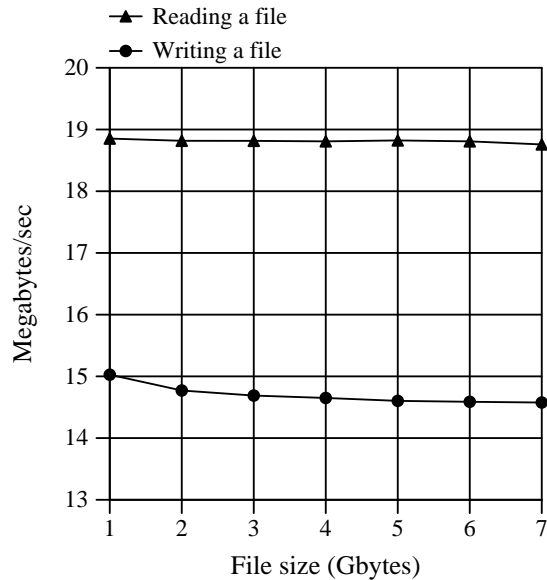
The results that follow were obtained on a CONVEX C3220 reading and writing to an eight-way striped filesystem with a total capacity of 7.4G. The read and write request sizes are held constant at 256K. Graph 1 gives the total time in seconds to read and write files ranging in size from 1G to 7G on a filesystem with a 64K block size. Graph 1 shows that the elapsed time increases linearly with file size and that reading a 7G file on a C3220 with a high performance disk stripe takes about six minutes. Writing a 7G file takes a little over eight minutes.

Graph 2 plots the read and write I/O rate in M/sec as the file size increases from 1G to 7G. The I/O rate for reads is a little less than 19M/sec and shows a slight decrease as the file size increases. The write rate shows a slightly larger decrease; at 1G the performance is 15M/sec and at 7G, the rate drops to 14.6M/sec. The largest decrease is between 1G and 2G. This corresponds to the move from single to double indirect blocks that occurs at 1.7G on a filesystem with a block size of 64K, indicating that this move has a slight impact on performance for large sequential writes. This effect is smaller than might be expected because the performance bottleneck for a high speed striped filesystem is the movement (copying) of data between buffers, and not the file system structure. Experiments with filesystems using unrealistically small filesystem block sizes (to force the use of triple indirect blocks) have shown that a similar slight decrease in performance can be expected when moving from double to triple indirect blocks.

¹⁰`stat64_t` is a typedef for the new kernel structure used during `stat64()` calls on large files. `stat_t` was also added for orthogonality.



Graph 1: Data Using Elapsed (Wall Clock Time)



Graph 2: Data Using Rates

Conclusion

We were able to limit the kernel changes required to support large files since the filesystem already had on-disk support for large files up to 1T-512 and ConvexOS already supported disk striping. Our major addition to ConvexOS was providing a second, 64-bit system call interface for key system calls. The most difficult decision we had to make was choosing the C programming model to implement. The model we choose was the simplest to implement from a kernel, compiler, and library standpoint while still meeting our customers' requirements. We were able to meet our primary goal of seamless FORTRAN support.

Our implementation of large files took approximately 45 programmer weeks to implement. This included:

- 20 weeks for utility work
- 18 weeks for kernel work
- 4 weeks for FORTRAN support
- 3 weeks for library work

In addition, 15 weeks were spent testing the kernel, utilities, and libraries. Our performance investigations show that the large file implementation had little effect on the I/O rates for large, sequential reads and writes.

Availability

Our implementation of large files is part of ConvexOS beginning with version 10.0. Source code is part of the standard OS source distribution and is available to ConvexOS source code licensees.

References

- [ANSI90] Accredited Standards Committee X3, Information Processing Systems, *American National Standard for Information Systems – Programming Language C*, X3 Secretariat: Computer and Business Equipment Manufacturers Association, Washington, DC, February 14, 1990.
- [CONVEX89-1] CONVEX Computer Corporation, *CONVEX POSIX Concepts*, Part Number 710-005030-000, Richardson, TX, December 1989.

- [CONVEX89-2] CONVEX Computer Corporation, *CONVEX POSIX Conformance*, Part Number 710-002030-200, Richardson, TX, December 1989.
- [CONVEX91] CONVEX Computer Corporation, *ConvexOS System Manager's Guide*, Order Numbers DSW-030 and DSW-031, Richardson, TX, November 1991.
- [IEEE88] The Institute of Electrical and Electronic Engineers, *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988*, The Institute of Electrical and Electronics Engineers, Inc, New York, NY, September 30, 1988.
- [Kernighan] Brian Kernighan, Dennis Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [Landherr91] Steve Landherr, "Directions in Disk Striping: The Virtual Volume Manager," Presented at Convex Users Group Conference, May 1991.
- [Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.
- [McKusick84] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, "A Fast File System for UNIX", Computer Systems Research Group, Department of EECS, Berkeley, CA, February 18, 1984.
- [NASA90] Jonathan Hahn, Bob Henderson, Ruth Iverson, Alan Poston, Tom Proett, Bill Ross, Mark Tangney, and Dave Tweten, *MSS-II External Reference Specification*, NAS Systems Division, NASA Ames Research Center, 1990.
- [Polk88] Jeff Polk, Rob Kolstad, "A Faster UNIX Dump Program," *USENIX Winter Conference Proceedings*, pp 125-129, February 1988.

Author Information

Dave Shaver is a kernel engineer at CONVEX and worked on the kernel and filesystem during the large file project. He is currently involved in OS design and implementation for future and existing CONVEX hardware platforms. He received his BSCS from Iowa State University. He can be reached electronically at shaver@convex.com.

Eric Schnoebelen is a utilities engineer at CONVEX. He received his BSCS from Iowa State University. He is responsible for new development and continuing support of the ConvexOS utilities and system libraries. Prior projects include the addition of POSIX.1 support into the libraries and utilities. Continuing projects include preliminary incorporation of some POSIX.2 functionality into ConvexOS. He can be reached electronically at schnoebe@convex.com or eric@cirr.com.

George Bier is an OS performance specialist at CONVEX. He is responsible for measuring, modeling and improving the performance of OS and networking products. Before joining CONVEX in 1990, he was a graduate student at the University of Wisconsin for seven years, receiving his MSCS in 1984. He received his BS in Computer Engineering in 1983 from Columbia University. He can be reached electronically at bier@convex.com.

All three authors can be reached by US mail at:

CONVEX Computer Corporation
PO Box 833851
Richardson, TX 75083-3851