Reprinted from *Proceedings of the JPL-SIGNUM Conference on the Computing Environment for Mathematical Software*, 17-18, July 1981, JPL Publication 81-67.

A Portable File System*

David R. Hanson

Department of Computer Science, The University of Arizona
Tucson, Arizona 85721

1. Introduction

Input/output is one of the most machine-dependent aspects of programming, especially for portable software. The large range of i/o and file system capabilities among existing computer systems makes it extremely difficult to avoid idiosyncratic problems in even the most carefully engineered portable systems. A typical solution to this dilemma is to use the 'standard' i/o and file primitives defined in the language in which the portable system is written [tan78]. In Fortran, for example, it is common practice to use only the forms of the i/o statements defined in the ANSI standard, and to use verifiers that aid in the detection of non-standard constructs [ryd74]. Another common approach is to define a small set of relatively low-level i/o routines that can be easily implemented and can model the capabilities of most commonly available file systems. By funnelling all i/o through these routines, portability problems are isolated in their machine-dependent implementation. The software described in [ker76] is evidence of the success of this approach.

A problem with these traditional approaches is they invariably sacrifice capability and efficiency for portability. Designs based on these approaches tend to have only the 'lowest common denominator' in capabilities of the intended host systems, such as sequential i/o to character files of restricted names. Enhancements may of course be added, but at the expense of an increase in implementation complexity and a reduction in portability.

The heart of the problem with traditional approaches to portable i/o systems lies in their attempt to manipulate highly machine-dependent objects—host machine files and file names. This paper describes a portable file system that makes files and their names machine-independent. The most important advantage of this approach is that i/o is not limited by the target systems. For example, capabilities such as random access, multiple access, and automatic expansion of files, which are absent in some commercial operating systems, are provided by the portable file system.

The portable file system—PFS for short—is the combination of a portable file directory system [han80a, han81] and a portable i/o system [han80b]. It provides machine-independent files and file names, a hierarchical directory structure in which to organize files, and a set of directory manipulation and i/o primitives. The directory structure and primitives are similar to the structure and primitives of the UNIX [rit74] file system. The PFS is, in large part, a portable implementation of the UNIX file system. It is packaged as a set of Ratfor [ker75] (and hence Fortran) functions and subroutines, which is loaded with the program or system that uses it. The implementation techniques are similar to those used in UNIX [tho78] and are described in [han80a] and [han80b].

2. Directories

The directory structure in the PFS is a rooted tree structure in which the leaves are files or directories and the nodes are directories. A directory is simply a list of files and directories. The root of the tree is denoted by /, and files and directories are denoted by their 'path', which specifies their absolute position in the tree.

A path is composed of the names of the nodes on the path from the root to the desired file or directory. The path components are separated by slashes, e.g. /source/pfs/alloc.rat. The names '.' and '..' refer, respectively, to the directory itself, and to its immediate ancestor. These names may be used as path components, providing a explicit means of using the structural properties of the tree. If a file name does not begin with /, it is taken to be rooted at the 'current directory'. For example, if the current directory is at /source/pfs, the name alloc.rat refers to /source/pfs/alloc.rat. Files and directories are equivalent with the exception that directories cannot be written by the user.

The primitives that deal exclusively with the directory structure are summarized in Table I.

Table I. PFS Directory Primitives

chdir(name)	change current directory to name
link(name1,name2)	make a link to name1 named name2
mkdir(name)	make a directory named name
rmdir(name)	remove directory named name
stat(name,array)	return information about file name

The current directory is changed by chdir. link establishes alternate names for a file. Directories are created by mkdir and, once empty, are deleted by rmdir. Information about a file (or directory), such as its size and date of creation, is returned by stat.

3. Primitives

A PFS *file* is similar to a file in UNIX and may be thought of as a finite sequence of characters or bytes. The PFS is insensitive to the range of byte values, so it can accommodate both 'binary' and 'character' files. Primitives are provided to create, delete, and open files, and to read and write characters anywhere within a file. Files are as large as is necessary to accommodate what is written to them, but are otherwise featureless. The basic primitives are summarized in Table II. Most primitives return a value indicating the success or failure of the operation.

Table II. PFS I/O Primitives

fd = fopen(name,mode)	open file name
fd = fcreate(name,mode)	create and open file name
fclose(fd)	close a file
n = fread(buffer,count,fd)	read from a file
n = fwrite(buffer,count,fd)	write to a file
pos = fpos(offset,type,fd)	position 'i/o pointer'
fremove(name)	delete file name

Existing files are opened for i/o by fopen. The argument name is the name of the file and mode is READ, WRITE, READWRITE, or APPEND and indicates 'how' the file is to be accessed. If the file exists, fopen returns a 'file descriptor', which may be thought of as a 'handle' that is used to access an opened file. Descriptors are similar in concept to channel numbers or Fortran unit numbers. Their values are never inspected explicitly but are passed to other primitives to indicate the opened file on which they should operate.

New files are created by fcreate, which creates the named file and opens it as if fopen had been called. If the file already exists, it is truncated to zero length and opened.

Opened files are closed by fclose(fd).

Data transfer to and from opened files is performed by fread and fwrite. fread reads up to count characters from the opened file indicated by the file descriptor fd into buffer. It returns the number of characters actually read, which may be 0 when the end of the file is reached. fwrite writes count characters from buffer to the file indicated by fd. Writing beyond the current size of the file is permitted, and the file is automatically extended to accommodate what is written to it. For symmetry with fread, fwrite returns the number of characters actually written.

An 'i/o pointer' is associated with each opened file and is advanced by fread and fwrite. It can be repositioned by fpos according to the values of offset and type. If type is 0, offset specifies a position relative to the beginning of the file; if type is 1, offset specifies a position relative to the end of the file; and if type is 2, offset specifies a position relative to the current positon of the file. fpos returns the previous position of the file.

Files are deleted by fremove. Deletion of opened files and files with aliases is permitted; the file is actually deleted upon the removal of the last reference to it. After deletion, all space occupied for the file is available for reuse.

4. Conclusions

The portable file system provides a machine-independent concept of file and i/o primitives that offer greater flexibility than is found in many operating systems. It is typically more efficient than the traditional approaches to portable i/o systems. For example, measurements on a DEC-10 and Cyber 175 show a 25-35 percent improvement over Fortran i/o for sequential character files.

Perhaps the best characterization of PFS is an abstract data type 'file'. It provides a data structure, file, and a set of operations on that structure. This characterization clarifies the important difference between the PFS approach and traditional approaches, which attempt to provide a set of operations on unspecified and highly machine-dependent data structures—host machine files.

References

[han80a]
Hanson, D. R. A Portable File Directory System. *Software—Practice & Experience 10*, 8 (Aug. 1980), 623-634.

[han80b]
Hanson, D. R. A Portable Input Output System. Tech. Report 79-17a, Dept. of Computer Science, The University of Arizona, Tucson, Nov. 1980.

[han81]
Hanson, D. R. Algorithm 568: PDS—A Portable Directory System. *ACM TOPLAS 3*, 2 (Apr. 1981), 162-167.

[ker75]
Kernighan, B. W. Ratfor—A Preprocessor for a Rational Fortran. *Software—Practice & Experience 5*, 4 (Dec. 1975), 396-406.

[ker76]
Kernighan, B. W. and Plauger, P. J. *Software Tools*. Addison-Wesley, Reading, MA, 1976.

[rit74]
Ritchie, D. M. and Thompson, K. The UNIX Timesharing System. *Comm. ACM 17*, 6 (Jul. 1974), 365-375.

[ryd74]
Ryder, B. G. The PFORT Verifier. *Software—Practice and Experience 4*, 4 (Dec. 1974), 359-377.

[tan78]
Tannenbaum, A. S., Klint, P. and Bohm, W. Guidelines for Program Portability. *Software—Practice and Experience 8*, 6 (Nov. 1978), 681-698.

[tho78]
Thompson, K. UNIX Implementation. *Bell System Tech. J. 57*, 6 (Jul. 1978), 1931-1946.

*This work was supported by the National Science Foundation under Grant MCS-7802545.