SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 95–100 (1983)

Short Communication

A PORTABLE INPUT/ OUTPUT SYSTEM

DAVID R. HANSON

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

Input and output are essential and often aggravating hurdles to be overcome in writing portable software. Idiosyncratic i/o can negate the portability of an otherwise portable system. Typical approaches to portable i/o rely on the definition of a small set of low-level i/o primitives that can be implemented easily using the 'middle-level' i/o facilities on the host computer. These approaches can result in i/o systems that are inefficient and lack richness, which limit their range of applicability. These conventional portable i/o systems manipulate machine-dependent files via a few machine-independent primitives. This note describes the design, implementation and application of a portable i/o system that manipulates *machine-independent* files via primitives that provide i/o capabilities atypical of many operating systems.

KEY WORDS Portability I/O systems UNIX Ratfor

INTRODUCTION

Input/output is one of the most machine-dependent aspects of programming and one of the more aggravating hurdles to be overcome in producing portable software. Indeed, since most portable systems rely on i/o to some extent, each is burdened with a serious *a priori* machine dependency. The range of i/o capabilities among computers is so large that the portability of even the most carefully engineered software can be thwarted by i/o idiosyncracies.

A typical solution to this dilemma is to use the 'standard' i/o primitives defined by the language in which the portable system is written.[1] For example, in Fortran it is common to use the i/o statements defined in the ANSI standard, and to use verifiers that detect of non-standard constructs.[2]

Another common approach, based on abstract machine modelling,[3] is to define a small set of relatively low-level i/o routines that can be implemented easily and can model common i/o devices,[4,5] Funnelling all i/o through these routines isolates portability problems in their machine-dependent implementation. The widespread use of the software described in Reference 6 attests to the success of this approach.

Despite the success of the abstract machine modelling approach, it suffers from several problems. To achieve portability, the i/o primitives are usually designed to accommodate the 'lowest common denominator' in i/o capabilities of the intended host systems. The primitives must necessarily exclude all but the most basic i/o capabilities, which limits their applicability and constricts the systems in which they are used. Most designs, for example, provide only sequential i/o involving character files. Enhancements may of course be added, but at the expense of implementation complexity or a reduction in the number of machines on which they can be realized. The i/o primitives used by the software in Reference 6 are a good example of this trade-off. Those primitives, drawn from UNIX,[7] provide more capabilities than are present in many host systems including a flexible random access scheme. The efficient implementation of these primitives is non-trivial, however, and typically requires approximately 6 man-months even on 'friendly' systems.[8]

Another problem is efficiency. The lowest common denominator approach often relies on the use of 'middle-level' i/o facilities such as a Fortran i/o system. This additional layer is required for portability reasons, but adds overhead. In addition, any semantic mismatch between the middle-level facilities and the portable i/o primitives they are used to implement results in an inefficient i/o system. The primitives in Reference 6 again provide an example: Their implementation using Fortran i/o is dreadfully slow. Any 'real' use of that software requires a finely tuned assembly language implementation.

The heart of the problem with traditional approaches to portable i/o systems is the attempt to manipulate highly machine-dependent objects—

0038–0644/83/010095–06$01.00

Received 27 April 1982
Revised 18 August 1982

host machine files. It is this restriction that necessitates the use of middle-level i/o systems, which turns the problem of manipulating machine-dependent files into one of interfacing with possibly machine-dependent middle-level i/o systems. Thus, to accommodate many computers, a portable system must be designed to rely on only the intersection of the capabilities of the various middle-level systems, i.e. the lowest common denominator.

This note describes a portable i/o ystem, PIOS, that avoids some of these limitations by making files themselves machine-independent, obviating the need for middle-level assistance. There are several advantages to this approach. First, and most importantly, the PIOS primitives are not limited by the capabilities of middle-level systems. For example, there is no reason to exclude capabilities such as random access i/o. Indeed, PIOS provides capabilities, such as random access, multiple access and automatic expansion of files, that are not found in some commercial systems. Second, the overhead of the middle-level routines and inefficiency induced by semantic mismatch are avoided. Last, as described below, the implementation of the PIOS primitives is often simplified because the file format can be adjusted to best suit the host computer.

THE INPUT/OUTPUT SYSTEM

On most systems, i/o primitives provide the means for accessing named files. Such primitives rely—at least conceptually—on two subsystems: a directory system and an i/o system. The distinction between these two systems is rarely made because they are usually combined as a single entity in existing operating systems. Portability constraints, however, focus attention on the distinction, and force a precise characterization of the dependencies involved in order to accommodate a wide range of hosts and applications.

Directory systems map *names* to *files*, and i/o systems operate on those files. The mapping from names to files can range from a very simple 1–1 mapping to more complex hierarchical many–1 mapping. Directory systems also provide other services such as access control. Adherence to this distinction between directory and i/o sybsystems simplifies their implementatoon and enhances flexibility.[9]

PIOS is an i/o subsystem. Its primitives operate on files, but not on names; it is independent of any specific directory system. It may be used with a machine-dependent or a portable directory system. Portable directory systems provide a machine-independent method for naming files. For example, the portable directory system PDS[10] supports a hierarchical directory structure, similar to that in UNIX and Multics.[11] and a set of primitives for manipulating that structure. Combining PIOS with potentially any directory system adds flexibility and increases applicability.

Since PIOS does not operate on names, it is not normally called by the user directly, but is called by applications-level primitives that first map names to files. One way to view PIOS is as a portable middle-level i/o system similar to those described above. An advantage of PIOS is that it is a known quantity—its usage does not change from computer to computer. In addition, it provides capabilities not normally found in middle-level systems such as Fortran i/o systems, and its implementation is designed to be more efficient than those systems.

A PIOS *file* is a finite sequence of charcters or bytes as in Multics[12] and UNIX. PIOS is insensitive to the range of byte values, so it can accommodate both 'binary' and 'character' files. Primitives create, delete and open files, and 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.

There are six PIOS primitives. Most return a value indicating the success or failure of the operation. Files are created by *createi*(*id*, *hostname*). As described below, PIOS uses host files in order to implement its concept of files; *hostname* is the name of that file. If the creation succeeds, *createi* places a character string identifying the new file in *id* and returns a success indication as its value. Details of *id* are described below.

The primitive *openi*(*id*, *file*) opens the file indicated by *id*. It returns in *file* a structure describing an opened file. The important two fields of the structure are *iop* and *size*. The value of the *size* field is the current size of the file in characters. The *iop* field contains an 'i/o pointer' whose value is used to indicate where in the file the next i/o transfer is to occur. The other fields are used internally by PIOS. The primitive *closei*(*file*) closes the opened file indicated by *file*.

Note that *openi* does not require an argument to indicate how the file is to be opened (e.g. 'read', 'write', 'append', etc.). At the PIOS level, files are simply opened, and may be read or written as desired. Constraints on the mode of access to an opened file must be enforced by the applications-level primitives. Doing so within PIOS would require *a priori* knowledge of what modes are meaningful for all applications.

The values of *id* returned by *createi* and used by *openi* are the range of the mapping function performed by the directory system. A typical implementation of an applications-level file creation routine, for example, calls the directory system to

map the name into an *id* and, if that succeeds, calls *openi* with the resulting *id* and a uninitialized *file* structure. The initialized *file* structure returned by *openi* is then used as an argument to other i/o primitives.

Data transfer to and from opened files is performed by *readi* and *writei*: *readi*(*buffer*, *count*, *file*) reads up to *count* characters from the opened file indicated by *file* into *buffer*, one character per word. Characters are read starting at the position indicated by the current value of *file.iop*, which is incremented by the number of characters read. *readi* returns the number of characters actually read, which may be 0 if *file.iop* is greater than or equal to *file.size*. Writing is similar to reading: *writei*(*buffer*, *count*, *file*) writes *count* characters from *buffer* to the file indicated by *file*. Writing starts at the position indicated by the current value of *file.iop*, which is incremented by the number of characters written. Unlike reading, however, writing beyond the current size of the file is permitted. In that case, the file is automatically extended to accommodate what is written to it, and its size is increased accordingly. Writing even apparently non-contiguous parts of a file is permitted. In this case, the *logical* size of the file is the largest value ever assumed by the i/o pointer. Unwritten parts of the file are not physically allocated unless necessary.

A file may be opened for reading more than once and each open is treated separately. A file may be opened for reading and writing at the same time, but the data read may depend on the data written. In practice, the potential for collision does not cause problems for simple applications. If it is a problem in more sophisticated applications, such as data base applications, checks to prohibit multiple access to a file opened for writing can be performed by the applications-level primitives. Indeed, the absence of such *a priori* restrictions is one of the advantages of PIOS and similar systems;[13] the user can enforce whatever constraints are appropriate for the application.

Files are deleted by *deletei*(*id*) where *id* is the identifying string returned by *createi*. Deletion of opened files 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.

As the above primitives illustrate, PIOS occupies a unique position in the continuum of i/o systems that range from the machine-independent to the machine-specific. It is generally less efficient than less portable systems and more efficient than more portable systems. For example, the applications-level primitives described below are typically less efficient, but only slightly more portable, when implemented with Fortran i/o than with PIOS. Moreover, the additional capabilities of PIOS offset the slight loss of portability.

IMPLEMENTATION

PIOS is a set of procedures, written in Ratfor[6, 14] (and hence Fortran), which is loaded with the program or system that uses it. The implementation relies on a few simple host machine i/o operations, facilitating implementation on a large number of machines.

The PIOS primitives are simple in concept, but their *direct* implementation can be very difficult on some systems. The random access aspect of *readi* and *writei* and the automatic extension of files often cause problems. The implementation techniques used in PIOS are based on those of Multics[12] and UNIX,[15] and are designed so that it can be implemented on even the most hostile systems, such as those in which files are fixed length and have a specific record structure. The basic technique is to spatially multiplex many PIOS files within one large host system file and access that file via four simple machine-dependent primitives.

The host file is assumed to be divided into a sequence of fixed-size blocks of characters (the actual size of the blocks is a compile-time parameter). In addition to open and close routines, machine-dependent routines to read and write arbitrary blocks are required. All i/o to the host file is in terms of complete blocks.*

PIOS files are identified by strings of the form '*-n hostname*' where *n* is the block number within the host file *hostname* that contains information necessary for accessing the file. It is this form of string that is returned by *createi* as the file *id*.

A file consists of data blocks and *information blocks* used for accessing the file. Whereas data blocks consist simply of *m* characters, where *m* is the block size, information blocks consist of *k* nonnegative integers, which are usually interpreted as block numbers. The precise value of *k* depends on the radix chosen for its representation, the block size, and largest value of *k* required, which must at least be the size of the host file in blocks. For example, using radix 256 in an implementation involving 512 8-bit bytes per block amounts to a binary representation. The initial implementation on the DEC-10 involving 640 7-bit characters per block used radix 10, which yields a character-oriented representation. Such representation de-

* The use of one large host file is inessential. On systems that discourage large files, the sequence of fixed-size blocks can be simulated by a number of small files, providing they can be accessed given a single name.

tails are not fixed by the implementation and can be tuned to the host computer.

The value of n in an *id* is the block number of the root information block for the file. Denoting an information block by the array $b[1:k]$, the root block has the following layout: $b[1]$ indicates the arrangement of the remaining information blocks; $b[2]$ and $b[3]$ give the size of the file in characters; and $b[4]$ to $b[k]$ contain block numbers of data or additional information blocks depending on the value of $b[1]$.

Addressing of data blocks is similar to the scheme used in UNIX and WFS.[13] If $b[1]$ is 0, $b[4]$ to $b[k]$ contain block numbers of data blocks, which accommodates 'small' files of up to $m\times(k-3)$ characters. If $b[1]$ is 1, $b[4]$ to $b[k]$ contain block numbers of additional addressing blocks, each of which contains k block numbers of data blocks, permitting 'large' files of up to $m\times k\times(k-3)$ characters. Information and data blocks are allocated automatically on demand. It is possible, for example, to have a 'holes' in a file.

Block allocation is handled by keeping a linked list of free blocks, much as is done in UNIX. The first block of the host file is used to store the head of this list along with the size in blocks of the host file.

PIOS handles up to a fixed number of host files simultaneously. It also maintains a software cache of the most recently accessed blocks. Including this machinery within PIOS permits very simple implementations of the machine-dependent primitives without a significant loss of efficiency. For example, on the DEC-10 the primitives occupy about a page of assembly language. These features do, however, require that extra care be taken to ensure that the data structures in the host file are kept up to date. PIOS updates these structures periodically depending on the modification frequency. In addition, *updatei* may be called to cause them to be updated, for example, after closing the last opened file.

APPLICATIONS

PIOS is useful in applications where a machine-independent representation of files and flexible i/o capabilities would enhance utility. It may be used with or without a directory system. Simple portable data base systems and self-contained language systems, such as APL, are examples of systems where PIOS could be used without a directory system. An implementation of a set of general-purpose tools, such as those described in Reference 6, on computers with limited i/o facilities is an example where an interface to a directory system would be necessary.

This latter usage is the typical use of PIOS. For example, the applications-level primitives used in Reference 6 have been implemented with a simple directory system that provides a 1–1 mapping from names to PIOS files. Those primitives, which amount to 65 lines of Ratfor, include routines for opening and closing files and reading and writing single characters. File creation is handled as a form of opening. Files are opened by *open*(*name*, *mode*) where *name* names the file, and *mode* is either READ or WRITE, indicating how the file is to be accessed. *open* returns a *file descriptor*, which is an index into a table of *file* structures and is used to access the opened file. Descriptors, similar to Fortran unit numbers, are never inspected explicitly; they are passed to other applications-level primitives to indicate the opened file on which they should operate. Files are closed by *close*(*fd*), which amounts to a call to *closei* and updating the table of *file* structures maintained by *open*.

Files are read, a character at a time, by *getch*(*c*,*fd*) where *fd* is the file descriptor returned by *open*. *getch* reads the next character and returns it in *c* and as the function value. It returns the distinguished character EOF at end of file. Files are written by *putch*(*c*,*fd*).

The implementation of *getch* and *putch* is a straightforward usage of *readi* and *writei*. The actual i/o being performed by *readi* and *writei* is done in units of blocks. PIOS permits applications-level primitives to be designed to accommodate the application with minimal concern about the efficiency of the actual i/o.

Table I gives transfer rates for these primitives implemented using three different i/o systems on two computers (operating systems are indicated in parenthesis). The rates are normalized to those of the Fortran i/o systems. The machine-specific i/o systems are implementations of the above primitives in assembly language. Implementation using PIOS and the Fortran i/o system required essentially the same effort (2 days).

Table I. I/O system transfer rates

i/o system	DEC-10 (TOPS-10)	Cyber-175 (NOS/BE)
Fortran i/o	1·00	1·00
Portable i/o	1·34	1·24
Machine-specific i/o	3·49	1·82

These measurements were obtained by implementing the primitives using each of the indicated i/o systems and using those implementations to copy a 70,000-byte file. A more favourable comparison for PIOS appears in other tests that exercise its cache, such as reading the same file a number of times, or use features that are difficult to realize with Fortran i/o, such as random access. In addition, proportionally larger transfer rates are obtained by reading and writing more than one character at a time. Such modifications are typically easier to make and are more effective in PIOS than in a Fortran-based i/o system.

As suggested earlier, the implementation of these primitives using PIOS is more efficient than their Fortran implementation and less efficient than their machine-specific implementation. As indicated by the data in Table I, the increase in efficiency of using PIOS is larger on computers for which there is a large difference in efficiency between Fortran and assembly language i/o. Fortran i/o on the Cyber-175, for example, is more efficient relative to assembly language i/o than on the DEC-10, and the efficiency advantage is correspondingly less. While there is less to be gained in terms of efficiency by using PIOS on the Cyber-175, PIOS offers capabilities not readily available on that computer.

Another, more complicated, example of the use of PIOS is in a set of primitives that use PDS as the directory system. That implementation provides capabilities similar to those in UNIX and mirrors most of the capabilities of PIOS at a higher level. Totalling about 180 lines of Ratfor, it has primitives for file creation, deletion and renaming, permits opening files for reading, writing, or both, and can read and write arbitrary sequences of characters in both sequential and random access modes.

The facilities of PDS and PIOS significantly simplified the implementation of this latter example. Using the subsystems permitted most of the design effort to be concentrated on functional capabilities. For example, the implementation in actual use, which totals 230 lines, was enhanced to provide an interface to two i/o systems, PIOS and the conventional DEC-10 file system. The result is a single set of applications-level primitives for accessing both DEC-10 and PIOS files. This enhancement was particularly useful for the editor described in Reference 6. It uses a random-access scratch file in addition to the sequential input and output files. Since random access is difficult for the standard DEC-10 file system, a PIOS scratch file was used. The editor thus uses host i/o facilities for the relatively simple sequential access to user-specified files and PIOS facilities for the more complex random access to the hidden scratch file.

CONCLUSIONS

PIOS demonstrates that portability need not limit the flexibility and efficiency of i/o systems. Perhaps more importantly, it also demonstrates that flexible i/o systems do not have to be complicated. It therefore serves not only as a usable piece of software, but as a model for the design of machine-dependent i/o systems.

UNIX was the starting point for the initial design of PIOS (and of PDS), and much of its influence appears in the end result. There are, however, significant differences induced by portability requirements. The most notable difference is the logical and physical separation of the directory system and the i/o system. This separation required an accurate characterization of what information is required by each subsystem. For example, the i/o subsystem needs location and size information about a file, but not access rights information; the reverse is the case for the directory subsystem. In most systems, UNIX being an example, such information is typically treated together. While this view of the information is adequate for machine-dependent systems, it is too restrictive for machine-independent systems that, to be useful, must accommodate a wide range of applications.

The other difference is an additional degree of freedom provided in PIOS. In UNIX, all of the files in a directory must reside in the same 'file system' as the directory, which corresponds to requiring all files in a directory to reside in the same host file in PIOS. Such restrictions do not apply to PIOS. More importantly, however, they *may* be enforced by the user of PIOS if desired.

Using separate directory and i/o systems also simplifies modifications since the two sybsystems are each simpler than their combination. For example, specialized protection mechanisms can be added to PDS without affecting PIOS.[10] Likewise, additional i/o capabilities can be added to PIOS with no impact on PDS. The major disadvantage of separate directory and i/o systems is that they may get out of synchronization due to either software faults or subversion. Experience to date and similar experience elsewhere[9, 13, 16] indicates that the separation does not induce additional problems and can actually aid in the repair of errors due to software faults. The safety and security of such systems relative to traditional systems has yet to be investigated, however.

An important contribution of PIOS is that it provides a machine-independent concept of 'file'. When integrating PIOS with other software, however, this contribution can be a disadvantage. Other software, a Fortran compiler for example, cannot read a PIOS file. As a result, a utility to copy files in and out of PIOS is required. This

problem is another reason why the application-level primitives for the DEC-10, described above, are interfaced to two i/o systems. While this solution is not difficult, it is aggravating and requires more effort in the design of such primitives than is desirable.

Another disadvantage is that PIOS is limited to a single user. Two users simultaneously updating the same host file will result in chaos. This problem is very machine-dependent, and any portable solution will require assumptions about host systems that may not be valid for all systems. However, the PIOS would suit the increasingly common single-user, personal systems, and it could serve as the basis of a centralized, multi-user i/o system.

ACKNOWLEDGEMENTS

The many comments and suggestions of Chris Fraser and of the referees resulted in improvements in both the portable i/o system and this presentation. Gregg Townsend and Steve Wampler provided valuable assistance and advice on the Cyber implementation. This work was supported by the U.S. National Science Foundation under Grant MCS–7802545.

REFERENCES

1. A. S. Tannenbaum, P. Klint and W. Bohm, 'Guidelines for program portability', *Software—Practice and Experience*, **8**, 681–698 (1978).
2. B. G. Ryder, 'The PFORT verifier', *Software—Practice and Experience*, **4**, 359–377 (1974).
3. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—a review and evaluation', *Software—Practice and Experience*, **2**, 107–136 (1972).
4. R. C. Dunn, '*Design of a higher-level language transport system*', Ph.D. Dissertation, University of Colorado, Boulder, 1975.
5. W. M. Waite, 'System interface', in *Software Portability, An Advanced Course* (Ed. P. J. Brown), Cambridge University Press, London, 1977.
6. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass., 1976.
7. D. M. Ritchie and K. Thompson, 'The UNIX time-sharing system', *Communications of the ACM*, 17, 365–375 (1974).
8. D. R. Hanson, '*Installing version 3 of the software tools*', *Tech. Rep. TR 81–23*, Department of Computer Science, University of Arizona. Tucson, 1981.
9. A. D. Birrel and R. M. Needham, 'A universal file server', *IEEE Transactions on Software Engineering*, **SE-6**, 450–453 (1980).
10. D. R. Hanson, 'A portable file directory system', *Software—Practice and Experience*, **10**, 623–634 (1980).
11. E. I. Organick, *The Multics System: An Examination of its Structure*, MIT Press, Cambridge, 1972.
12. R. J. Feiertag and E. I. Organick, 'The Multics input-output system', *Proceedings of the Third Symposium on Operating Systems*, 35–41 (1971).
13. D. Swinehart, G. McDaniel and D. R. Boggs, 'WFS: a simple shared file system for a distributed environment', *Proceedings of the Seventh Symposium on Operating Systems*, 9–17 (1979).
14. B. W. Kernighan, 'Ratfor—a preprocessor for a rational Fortran', *Software—Practice and Experience*, **4**, 396–406 (1975).
15. K. Thompson, 'UNIX implementation', *Bell System Technical Journal*, **57**, 1931–1946 (1978).
16. B. W. Lampson and R. F. Sproull, 'An open operating system for a single-user machine', *Proceedings of the Seventh Symposium on Operating Systems*, 98–105 (1979).