SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 623–634 (1980)

A Portable File Directory System*

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

SUMMARY

A portable file directory system that provides a machine-independent method for specifying files is described. The portable directory system, or PDS, facilitates uniform usage of portable software by supplying a standard for file specification. The PDS supports a hierarchical directory structure, similar to that in UNIX and Multics, and a set of primitives for manipulating the structure and preparing files for input/output. While the PDS participates in file specification, it does not participate in the actual i/o; it has no impact on i/o efficiency. The implementation, which is easily extended to include additional capabilities, is described. Experience in using the PDS as a part of command interpreters and preprocessors is also discussed.

KEY WORDS Portability Adaptability File systems UNIX Modularity Ratfor

INTRODUCTION

Portability is a major concern of software engineering.[1,2] A program or system that can be easily moved from computer to computer greatly reduces the cost of that program on a per computer basis. Portability has been a major aspect of several programming languages such as SNOBOL4,[3–5] BCPL[6,7] and, more recently, C.[8,9] In addition, recent attempts have been made to make complete operating systems portable,[9,10] although the concept of 'portability' is slightly different in those cases.

While portability makes a 'standard' implementation of a language or system, say SNOBOL4, available on many machines, it does not guarantee that the *use* of that system will be in any way similar on those machines.[2] Differences in i/o capabilities, operating system peculiarities, character set differences, and local conventions all conspire to make the use of a portable system effectively machine-dependent. This effect is part of the motivation to porting an entire operating system; the use of the system is also ported in that case.[9,11]

Each computer has its own concept of 'file' and 'file directory'. Since files are the usual method of communication with programs, the capabilities of a file system can greatly influence—even pervade—the use of portable software. For example, the documentation of a software system must refer to the interaction with the computer's file system. On some computers, the influence is minimal; this is usually the case on computers with good file systems. On computers with inadequate file systems, however, the poor file system may be the limiting factor on the usefulness of portable software.

* This work was supported by the National Science Foundation under grant MCS78–02545.

0038–0644/80/0810–0623$01.00 *Received 10 October 1979*
© 1980 by John Wiley & Sons, Ltd.

This paper describes a portable file directory system that provides a machine-independent method of file specification. The portable directory system—PDS for short—provides capabilities far beyond those provided by many vendor-supplied systems. In addition, it is easily extended to accommodate additional capabilities, such as specialized protection schemes or information concerning particular usage of files. The PDS could, for example, be extended to perform some of the functions of a source code control system.[12]

The basic technique used in the design of the PDS is the separation of the information describing a file from the file itself. The PDS deals only with the former; it does not use or manipulate, in any way, the actual files. The importance of this approach is that the PDS is used to specify a file but does not participate in the actual i/o to that file. As a result, there is no impact on i/o efficiency when the PDS is used. In the simplest terms, the PDS provides a *mapping* from machine-independent file names to machine-dependent names. Use of the PDS implies use of a machine-independent directory system to locate files, and a machine-dependent i/o system to access them.

The PDS is packaged as a set of procedures, written in Ratfor,[13, 14] which is loaded with the program or system that uses it. A typical use of the PDS would be in programs that make heavy use of named files, especially in cases where a machine-independent means of naming files and a flexible directory structure would enhance the usefulness of the program. Another use would be in a set of tools, such as those described in Reference 14, implemented on computers with limited file systems. The PDS was originally implemented on a DEC–10 and has been successfully ported to a Cyber–175.

The next section describes the directory structure provided by the PDS and the primitives that operate on that structure. Implementation of the system and experience in using the PDS are covered in subsequent sections.

DIRECTORY STRUCTURE AND PDS PRIMITIVES

To be effective, the PDS must provide a useful directory structure and a set of primitives for manipulating that structure. There are a vast number of existing file systems; one of the more useful is the hierarchical system provided by UNIX[15] and Multics.[16] The tree structure of the UNIX file system is the basis for the directory structure implemented in the PDS, and many of the primitives are similar to UNIX primitives.

Directories

The PDS provides 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. An example is shown in Figure 1; directories are indicated by circles, files by squares. The root of the tree is denoted by '/'. 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. the path for file 'f8' in Figure 1 is '/d8/d5/d6/f8'. As far as the PDS primitives are concerned, files and directories are equivalent with the exception that directories are manipulated by the PDS and files are not. Thus '/d1/d4' specifies the directory 'd4', which can be read like any other file.

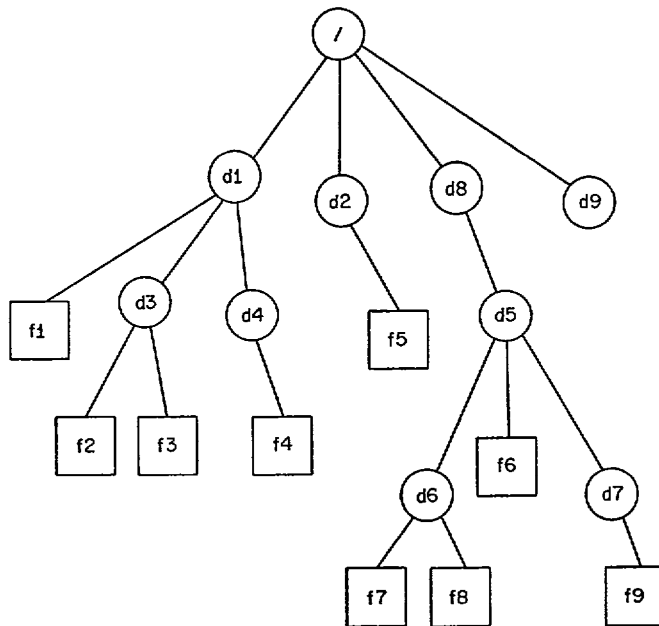

Figure 1. A directory structure

Each directory contains at least two entries: '.' refers to the directory itself and '..' refers to the direct ancestor of the directory. The direct ancestor of the root is the root (Figure 2). Both '.' and '..' are file names in the ordinary sense and may be used in paths just like any other directory. Thus both '/x/file1' and '/x/y/../ ./file1' refer to 'file1' in Figure 2.

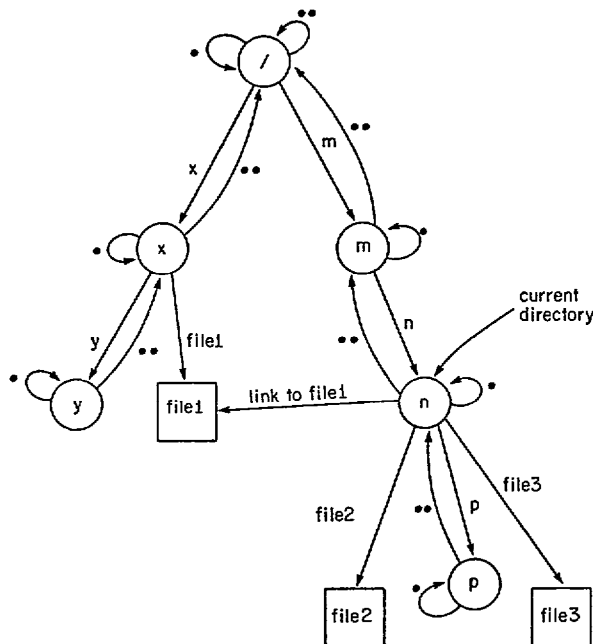

Figure 2. Links and dot conventions

The notion of a 'current directory' is also supported. For example, in Figure 2, the current directory is 'n'. If a path does not begin with '/', it is taken to be rooted at the current directory. For example, with the current directory at 'n', the path 'file3' is equivalent to '/m/n/file3'. As another example, the path '../../x/file1' refers to '/x/file1'.

The use of a full path to refer frequently to files in directories other than the current directory may be tedious. This can be avoided by using 'links'. For example, in Figure 2 a link can be made to '/x/file1' from directory 'n'. Thereafter, when the current directory is 'n', 'file1' may be referred to simply as 'file1' instead of the long path given above. The name of the link need not be the same as the name of the file to which the link is directed.

The term 'link' is somewhat misleading in that, once established, the directory entry for a link is exactly the same as the directory entry for the file to which the link is directed. Links simply provide a means for a file to appear in more than one directory. Links to directories are not permitted, however. This restriction is imposed in order to simplify the allocation and deallocation of nodes by maintaining a tree structure. Only the PDS can create links to directories in response to specific primitives.

As mentioned above, the PDS does not manipulate actual files. The files depicted in Figures 1 and 2 simply 'contain' the machine-dependent names of the actual files. These machine-dependent names are referred to as *host names*. The basic function of the PDS is to map paths into host names.

Primitives

The primitives supported by the PDS fall into two categories: those that manipulate the directory structure, and those that map a path to a host name and perform some machine-dependent operation on the actual file. The latter kinds of primitives are mainly concerned with preparing files for i/o; none actually do any i/o.

The most basic of the directory manipulation primitives is

mkfile(*path, hname*)

which establishes *path* as equivalent to the host name *hname*. Subsequent references to *path* will be mapped into *hname*. For example,

mkfile(/sys/fc', 'sys:fortran')

might be used so that 'fc' in the 'sys' directory refers to the standard Fortran compiler. In practice, explicit use of **mkfile** is rare. Most mappings are established implicitly by the primitive **creatf** described below, which ultimately calls **mkfile**.

Directories are created by

mkdir(*path*)

which also makes entries for '.' and '..'. The current directory is set by

chdir(*path*)

For example,

chdir('/m/n')

sets the current directory to 'n' as shown in Figure 2;

chdir('..')

changes it to 'm'.

Most primitives return an error code if the requested operation is illegal or meaningless. For example, the argument to **chdir** must be a path to a directory.

Links are created by

link(*path1*, *path2*)

A link named *path1* is established to *path2*. For example, in Figure 2, if the current directory is 'n', the link to '/x/file1' could have been made by

link('file1', '/x/file1')

or

link('file1', '../../x/file1')

Links are removed by

unlink(*path*)

Every file has at least one link—the directory entry for that file. When all links to a file are removed, it is inaccessible and is removed from the PDS. Thus, **unlink** is used to perform the traditional 'delete file' operation found in many file systems. Whether or not the host file is actually deleted is determined by the host machine interface to the PDS.

unlink is also used to 'rename' files. Suppose the current directory is 'n', as shown in Figure 2, and that 'file3' is to be renamed to 'program'. This is accomplished by

link('program', 'file3')
unlink('file3')

The invocation of **link** will return an error if 'program' already exists in the current directory. This can be avoided, at the cost of losing 'program' if its exists, by preceding the call to **link** by

unlink('program')

unlink is also used to unlink directories, but directories must be empty, that is, contain only the entries '.' and '..'.

The primitives that deal in some way with actual host files are those that open and create files. The primitives are modelled after those in UNIX (see also Reference 14). Files are opened for i/o by

openf(*path*, *mode*)

openf maps *path* into the appropriate host name (the path must exist) and calls a machine-dependent routine to actually open the file. *mode* is passed untouched to the machine-dependent routine; it typically indicates how the file is to be opened (e.g. 'read', 'write', 'append', 'readwrite' etc.). Since *mode* is not modified by **openf**, it can be whatever is most appropriate on the host system. The PDS is said to be *transparent* to those arguments and functions that it passes along to machine-dependent routines.

openf returns whatever the machine-dependent open routine returns; a channel number or Fortran unit number is typical. It is useful, but not mandatory, to return distinguished values to indicate errors. The value returned by **openf** should be whatever is most useful as an argument in calling machine-dependent routines that perform the actual i/o (e.g. **read** and **write**).

There are *no* other i/o primitives in the PDS; it is completely independent of the actual i/o. Thus, while additional overhead due to the PDS is incurred in opening a file, none is incurred in accessing the file. The former is performed much less frequently than the latter, minimizing the effect of any additional overhead. A major contribution of the PDS is the provision of a hierarchical directory system with absolutely no time impact on i/o efficiency.

The primitive

creatf(*path, mode*)

is similar to **openf**, except that *path* must not exist. The path is created along with a host name and file. The generation of the host name is performed by **mkfile** with a null second argument:

mkfile(*path,* '')

indicates that a host name is to be generated. By default, this name is simply a unique integer; other conventions can easily be used instead. After creating the file, **creatf** opens it according to *mode* as if

openf(*path, mode*)

had been called.

creatf is the normal way by which mappings are entered in the PDS. Note that **creatf** changes the directory structure whereas **openf** simply examines it; the inclusion of **creatf** and **openf** was motivated in part by this separation of function.

The particular choice of **creatf** and **openf** is not fundamental to the PDS, but the notion of transparency is. On some systems (such as the DEC-10), opening a non-existent file for writing implies its creation. While modifying the PDS to accommodate this variation is straightforward, it does require that **openf** inspect the mode. The resulting loss in transparency is not disastrous in this case, but is symptomatic of a trend to be avoided; further changes that would require the PDS to participate in the actual i/o would negate its usefulness. Fortunately, experience to date suggests that **creatf** and **openf** provide *more* than is available on most systems. Typically, a subset of their full capabilities is used that does not compromise the transparency of the PDS.

There are several other primitives that deal with the implementation of the PDS; these are described in the next section.

IMPLEMENTATION

The implementation of the PDS is based partly on the implementation of the UNIX file system[17] and partly on the requirement that it use only simple sequential i/o (i.e. Fortran i/o) for portability reasons. The system consists of about 680 lines of Ratfor.[13] Ratfor was used not only for portability reasons, but because Fortran is typically the only widely available high-level language to which calls from other languages (e.g. PL/I and Cobol) can be made.

The PDS manipulates directories. In addition, however, it must permit those directories to be read as ordinary files. For example

openf('/','read')

might be used to open the root directory for reading. This capability is necessary in order to write a utility to list a directory, for example. Thus, with a few minor exceptions, directories are treated like ordinary files.

Each file (and directory) in the PDS is associated with an *i-node* (the term comes from UNIX). An i-node contains all of the information, including the host names, concerning the file. Specifically, an i-node contains

the file type ('d' for directory, 'p' for plain)
number of links to the file
creation time (integer of the form hhmm)
creation date (integer of the form mmdd)
creation year (integer of the form yyyy)
time of most recent access via the PDS
date of most recent access via the PDS
year of most recent access via the PDS
host name

For example, the i-node for the file created by

mkfile('/sys/fc', 'sys : fortran')

might be

```
p  1 1140  207 1979 1140  207 1979 sys:fortran
```

The i-node for a directory is similar except that 'p' is replaced by 'd'.

I-nodes are kept in a single file called the *i-list*. I-node n is line n in the i-list file. The i-list is stored in character format so that it can be read easily with Fortran i/o or its equivalent. For example, the i-list for the system depicted in Figure 2 is as follows.

```
d  4 1411  613 1979 1411  613 1979 root
d  3 1411  613 1979 1411  613 1979 2
d  3 1411  613 1979 1411  613 1979 3
d  2 1412  613 1979 1412  613 1979 4
p  2 1412  613 1979 1412  613 1979 host-file1
d  3 1414  613 1979 1414  613 1979 6
p  1 1418  613 1979 1418  613 1979 host-file2
d  2 1418  613 1979 1418  613 1979 8
p  1 1418  613 1979 1418  613 1979 host-file3
```

I-node n is located by simply reading the i-list up to line n. More efficient techniques are possible using different i-list formats, but this is usually unnecessary for two reasons. First, the time involved in locating an i-node is usually in response to an **openf** call whose execution time is not critical. Second, and more importantly, the PDS maintains a software cache of the most recently referenced i-nodes; the result is that the i-list is read infrequently.

The i-list provides the basic mapping mechanism used by the PDS. It maps i-node number, or i-number, to a host name. The implementation of directories is built on this mechanism. A directory is a file of lines, each line is one entry containing

i-number   PDS name

Directories provide a map of PDS name to i-number. In conjunction with the i-list, the map from PDS name to host name is obtained.

For example, the directory for '/' in Figure 2 is

```
1  .
1  ..
2  x
3  m
```

The directory for '/x' is

```
2  .
1  ..
4  y
5  file1
```

and the directory for '/m/n' is

```
6  .
3  ..
5  file1
7  file2
8  p
9  file3
```

Note the implementation of '.' and '..'. Since they are simply file names, no special handling is required to scan paths in which they appear. It is, however, necessary to forbid unlinking of '.' and '..'.

The structure of a PDS name is arbitrary. In the current version, almost anything is acceptable in a name, e.g. upper- and lower-case letters are treated as distinct. One of the advantages of this handling of names is that naming conventions can be changed as desired to suit the best application.

The additional level of indirection introduced by separating directories from the i-list makes the implementation of links trivial. As shown above, the entry for '/m/n/file1' is identical to '/x/file1', and the i-node for that file has a link count of 2. Note that the name of the link does not have to be the same as the file linked to; names are a part of directories only.

This level of indirection also facilitates the implementation of the hierarchical structure. A directory entry for a directory is no different from an entry for a file; the characteristics of the file are in the i-node, not the directory entry. Note that in order to forbid links to directories and to implement calls such as **chdir**, it is necessary to read the i-list. The i-list is, in fact, the focus of attention in the PDS. This motivates the use of the cache.

There are three PDS primitives that concern initialization and updating of directories and the i-list.

mkds(*root, ilist*)

creates a directory system by creating and initializing a root directory, which is always i-node number 1, and an i-list. The arguments specify the host names of the root and i-list, respectively; null arguments result in the use of 'root' and 'ilist'.

chds(*root, ilist*)

is used to change to a PDS with another root and i-list. When used in conjunction with **mkds**, **chds** facilitates the use of more than one PDS in the same program. The arguments, which specify the host names of the root and i-list for the other PDS, default to the current root and i-list names, respectively.

sync( )

causes the in-core copy of the current directory and the i-node cache to be written to the appropriate files. In the case of the i-node cache, the entire i-list is read, merging the in-core i-nodes as necessary. This technique permits the i-list to be manipulated using sequential i/o. The PDS automatically performs a **sync** periodically.

Finally,

stat(*path, array*)

returns the i-node for *path* in *array*. This primitive is typically used to determine the existence of a file and to obtain the host name.

EXPERIENCE

The major uses of the PDS have been as a part of portable command interpreters and preprocessors, as an aid in describing the organization of large software systems, and as a part of various tools that benefit by having a more flexible directory system than is provided by the host system.

A command interpreter was written originally to test the PDS itself. Its command repertoire consists mainly of commands corresponding to PDS primitives or combinations of them. Examples are **mkfile**, **mkdir**, **chdir** and **link**, which invoke the PDS primitives of the same name. Other, more complex, commands include **rm**, which removes files, **rmdir**, which removes directories, **ls**, which lists the contents of directories, **mv**, which renames files and **pwd**, which prints the full path name of the current directory.

After the PDS was operational, however, this interpreter continued to be useful in its own right. In particular, it is a useful tool for describing the organization of large software systems, most notably the PDS itself and Icon.[18] The PDS permits the components of a system to be named in a machine-independent fashion and organized in a hierarchical structure. These capabilities are frequently sufficient to make clear the relationships among system components such as include files, common data structure definitions and procedures.

The Icon system is a case in point. Icon is a new programming language intended primarily for non-numeric applications. The Icon system consists of two major parts, a translator and a run-time system, composed of approximately 330 components, most of which are procedures. The directory structure provided by the PDS has proved useful for organizing the components of the Icon system so that the machine dependencies and subsystems are clearly delineated. After working with a system of Icon's size on a computer with a file system having essentially a flat structure, the utility of this sort of organizational capability is clear. It is, however, usually taken for granted on systems, like UNIX, with hierarchical file systems. The PDS provides a means for realizing this kind of structure on any computer.

Another use of the PDS is in a command preprocessor that translates machine-independent operating system commands in which files are referred to by their PDS name to the appropriate host command sequence. Commands have the form

name arg1 arg2 ... argn

Arguments are separated by blanks. If the name is a known command, the appropriate host command sequence is generated and, depending on the implementation, executed. For example, assuming the current directory is '/source/pds', the command

rc inode.r

specifies Ratfor compilation, which involves running the Ratfor preprocessor and the Fortran compiler. On the DEC–10 version, the command preprocessor obtains the host name for 'inode.r' (the full path is '/source/pds/inode.r'), creates 'inode.o' (the object file), and generates the following DEC–10 command sequence (assuming the host name for 'inode.r' is 'inode.rat').

```
.r ratfor
* inode.f10 = inode.rat
.r fortran
* inode.rel = inode.f10
.delete inode.f10
```

The command preprocessor obtains the host name for 'inode.r' using **stat** and uses **mkfile** to create 'inode.o' specifying 'inode.rel' as the host name. Other capabilities include commands that invoke editors, construct object libraries, and build executable programs. Commands that invoke PDS primitives, similar to those provided by the command interpreter described above, are also included.

If the command name is not recognized, it is assumed to be the PDS name of a program file. The arguments are assumed to be PDS names; they are mapped to the corresponding host names and the program is invoked with the mapped arguments. There is an escape mechanism to suppress the mapping on a per-argument basis.

The command preprocessor provides a set of machine-independent commands that can be used not only to describe the installation of software systems but to generate the appropriate host commands for doing so. The advantage of using the PDS is that it adds another degree of machine independence, specifically in the naming of files. Moreover, the command repertoire can be designed to take advantage of the hierarchical structure instead of being restricted by the lowest common denominator in file system capabilities. The result is not only a more portable command preprocessor, but a more flexible one, which is precisely what is needed in describing the installation of large complex software systems.

CONCLUSIONS

In retrospect, the design and implementation of the PDS is simple—once the design is clear. This substantiates Ritchie's claim[19] that there is really no excuse for not implementing a hierarchical file system as a part of any operating system.

The PDS demonstrates one approach to separation of function in a file system implementation. Indeed, it is the particular lines of separation chosen—separating

directory manipulation from file access—that permit the system to be portable. It is also what makes the PDS useful; the simple implementation facilitated the inclusion of capabilities beyond those supplied by many host systems. It also makes extensions, such as a simple protection mechanism, straightforward. Most importantly, more exotic capabilities such as automatic dependency relations[20] can be added to the PDS, whereas the complexity of these kinds of extensions would normally preclude their inclusion in traditional file systems.

One disadvantage of the PDS appears in interfacing its use with existing software; the Fortran compiler will not recognize PDS names. A command preprocessor, like the one described above, mitigates most of these difficulties, but its use is sometimes cumbersome. The preprocessor must be explicitly aware of which commands create files, which read files and which arguments refer to which kind of file usage. It must also be aware of programs that implicitly create files, like a listing and create the appropriate PDS names to refer to those files. While these problems are not insurmountable, they are aggravating and prevent the command preprocessor from being implemented by a general-purpose macroprocessor. In the command preprocessor described above, the generation of host commands is done in an *ad hoc* fashion. Current efforts are directed towards simplifying and automating this process. The ultimate goal is to provide a means of specifying the meaning of preprocessor commands in terms that will permit a command preprocessor for a specific host computer to be generated automatically.

Another disadvantage is that the PDS is limited to a single user at any one time. Unless some special action is taken, two users simultaneously updating the i-list or a directory may result in chaos. The major aspect of this problem is that it is machine-dependent. Any portable solution will require assumptions about host file systems that may not be valid for some systems. As an example of an alternative approach, suppose the host system has a method for restricting the access to a file to a single user. The i-list can be implemented as a group of small files, one for each i-node. Accessing an i-node would prevent other users from accessing it until the operation was completed. Not only is this approach wasteful of space on most systems, but it makes assumptions that may be incompatible with them. Effective multi-user access to the PDS requires that the PDS be centralized so that all users are being serviced by one PDS.

In practice, the advantages of the PDS outweigh the disadvantages. The additional capabilities provided by the PDS are attractive enough to encourage its use on computers with poor file systems. On the speculation that similar comments apply to i/o in general, research is currently underway to develop a portable i/o system that is independent of, but works in concert with, the PDS. The idea is the same as that for the PDS; the set of i/o primitives will be better than what is available on some systems. Initial results indicate that the i/o system is also easy to implement once the 'right' structure is discovered, but that it may have a more limited range of applicability. This is because only programs that use the portable i/o system can read those files. The important point about the PDS is that, once the PDS to host name mapping is performed, any program can read the actual file without additional overhead.

ACKNOWLEDGEMENTS

Chris Fraser assisted in the implementation of the command preprocessor and provided invaluable insight on the use of the PDS. He and Ralph Griswold made valuable suggestions concerning this presentation.

REFERENCES

1. P. J. Brown (Ed.), *Software Portability, An Advanced Course,* Cambridge University Press, London, 1977.
2. P. C. Poole and M. M. Waite, 'Portability and adaptability', in *Software Engineering, An Advanced Course,* F. L. Bauer (Ed.), Springer-Verlag, New York, 1975, pp. 183–276.
3. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language,* 2nd edn., Prentice-Hall, Englewood Cliffs, N.J., 1971.
4. R. E. Griswold, *The Macro Implementation of SNOBOL4, A Case Study of Machine-Independent Software Development,* W. H. Freeman, San Francisco, 1972.
5. R. B. K. Dewar and A. P. McCann, 'MACRO SPITBOL—a SNOBOL4 compiler', *Software—Practice and Experience,* **7**, 95–113 (1977).
6. M. Richards, 'BCPL: a tool for compiler writing and system programming', *Proceedings AFIPS Spring Joint Computer Conference,* **34**, 557–566 (1969).
7. M. Richards, 'The portability of the BCPL compiler', *Software—Practice and Experience,* **1**, 135–146 (1971).
8. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1978.
9. S. C. Johnson and D. M. Ritchie, 'Portability of C programs and the UNIX system', *Bell Syst. Tech. J.* **57**, 2021–2048 (1978).
10. D. R. Cheriton *et al.,* 'Thoth, a portable real-time operating system', *Comm. ACM,* **22**, 105–115 (1979).
11. B. W. Kernighan and J. R. Mashey, 'The UNIX™ programming environment', *Software—Practice and Experience,* **9**, 1–15 (1979).
12. M. J. Rochkind, 'The source code control system', *IEEE Trans. Software Eng.,* **SE-1**, 364–370 (1975).
13. B. W. Kernighan, 'Ratfor—A preprocessor for a rational Fortran', *Software—Practice and Experience,* **5**, 395–406 (1975).
14. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison–Wesley, Reading, Pennsylvania, 1976.
15. D. M. Ritchie and K. Thompson, 'The UNIX timesharing system', *Comm. ACM,* **17**, 365–375 (1974).
16. E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press, Cambridge, Mass., 1972.
17. K. Thompson, 'UNIX implementation', *Bell Syst. Tech. J.* **57**, 1931–1946 (1978).
18. R. E. Griswold, D. R. Hanson and J. T. Korb, 'The Icon programming language: an overview', *SIGPLAN Notices,* **14**, 18–31 (1979).
19. D. M. Ritchie, 'A retrospective', *Bell Syst. Tech. J.* **57**, 1947–1969 (1978).
20. S. I. Feldman, 'Make—A program for maintaining computer programs', *Software—Practice and Experience,* **9**, 255–265 (1979).