A Virtual Operating System

Dennis E. Hall, Deborah K. Scherrer, and Joseph S. Sventek
Lawrence Berkeley Laboratory

One complication you probably have no control over is your local computing environment. But even if it's horrible, as many are, you don't have to suffer stoically. Even a modest improvement of frequently used parts, like your programming and job control languages, is well worthwhile, and there's no excuse for not trying to conceal the worst aspects.

—Kernighan and Plauger, *Software Tools*

SUMMARY: Moving to a new system is costly and error-prone. The problem can be reduced through use of a virtual operating system that disentangles computing environments from their underlying operating systems. The authors report on their successful experience in doing this and achieving inter-system uniformity at all three levels of user interface: virtual machine, utilities, and command language.

1. Introduction

Associated with each computer system is a "local computing environment" or operating system interface. Today's computer marketplace offers a variety of such environments, each inextricably entwined with its own peculiar set of hardware components. Because of this, customers acquiring a new system must usually spend considerable time and effort moving both software and people to a new computing environment.

Under present conditions, even estimating the organizational impact of such a move can be extremely difficult. As a rule, moving to a new system is costly and error-prone. Therefore, many organizations have elected to stay with a single computer vendor in spite of an increasingly competitive hardware marketplace.

Although computer manufacturers have been effective in developing highly reliable operating systems, their computing environments are not usually examples of good human engineering. Customers, in an effort to minimize the cost of moving to new systems, have insisted that vendors remain compatible with historical precedent. This tends to discourage the removal of poor interfaces and inhibits the development of improved ones. As a result, bad interfaces seem to live on forever.

For many computer users there is no need to distinguish between the interface to an operating system and the operating system itself. We will show that under certain conditions a uniform system interface can be provided across machine boundaries without disturbing vendor software. The method consists of creating a virtual operating system.

Key words and phrases: computing environments, operating systems, virtual machines, system utilities, command languages, functional equivalence of operating systems, user mobility, user interface, moving costs

CR Categories: 4.35, 4.40, 4.6

This work was supported by the Applied Mathematical Sciences Program of the Office of Energy Research, of the U.S. Department of Energy under contract W-7405-ENG-48.

Reprints are available from the Lawrence Berkeley Laboratory as LBL report 10677.

Author's address: D.E. Hall, D.K. Scherrer, and J.S. Sventek, Lawrence Berkeley Laboratory, University of California, Berkeley, CA 94720.

2. The Virtual Operating System Approach

A real operating system presents three principal interfaces to its users [6]: the *virtual machine* or operating system primitives accessible through programming languages; the *utility programs* such as compilers, linkers, and editors, and the *command language* or means by which users access system resources from a terminal. Most system services are available through one or more of these interfaces (see Figure 1).

The idea of a virtual operating system is to provide standard versions of these interfaces, based on organizational requirements. Possible applications include data management environments, office information environments, real-time process control environments, and program development environments, to name a few.

Once the three interfaces are specified, implementation consists of:

- choosing one or more programming languages;
- developing run-time libraries or extending the selected programming languages to support the chosen virtual machine on each target system;
- implementing the utilities and command language in one or more of the selected programming languages, relying on the virtual machine to interface to the target operating systems;
- writing the necessary documentation.

A virtual operating system becomes a real operating system when the associated virtual machine corresponds to a physical machine. However, the emphasis in building a virtual operating system is on the interface presented to the user. The virtual machine is a highly idealized set of primitive functions geared to organizational programming requirements. It bears almost no functional resemblance to the underlying hardware which actually performs the work. In general, a virtual operating system is restricted to those parts of an ordinary operating system which an organization considers important in completing its work. Obviously, a single real operating system can support many virtual operating systems.

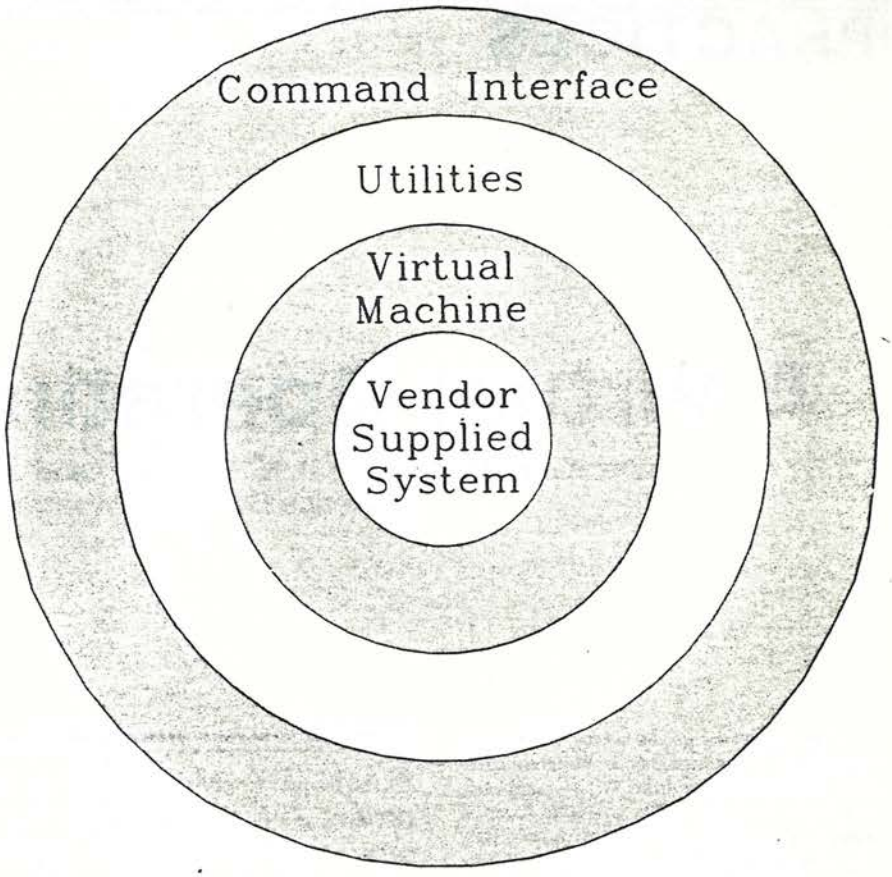

Fig. 1. A virtual operating system provides standardized versions of the three outermost system layers. Installation consists of interfacing the standardized virtual machine to the vendor supplied system.

To achieve the full benefits of this approach, the virtual machine must be implementable without changing the vendor software. This implies a functional equivalence between the chosen virtual machine and the target systems. A bootstrapping design procedure is therefore required. Every candidate virtual machine function must be tested on each target system before it can be adopted.

The virtual operating system approach reduces the problem of moving to a new system to the (nontrivial) problem of implementing a virtual machine. All utilities and user programs are completely portable since their interface to any particular operating system is through the virtual machine. Similarly, higher level procedures written for a portable utility are themselves portable. For example, a file containing editor commands will work on any machine supporting the editor utility. Finally, command language procedures are also portable, since the command language program is portable. The availability of the entire virtual operating system (virtual machine, utilities, and command language) makes it easy for users and programs to move from one vendor system to another.

We emphasize that this approach reduces the cost of moving both people and software to zero. The overhead is the cost of implementing the virtual machine on the candidate sys-

tem. This can be estimated by any knowledgeable system programmer and is completely independent of the number of people and the amount of software to be moved.

3. When is a Virtual Operating System Approach Desirable?

The advantages and disadvantages of a virtual operating system are much the same as those for a real operating system. However, the effort to develop and maintain a virtual operating system is usually far less than that expended for a real operating system: The most difficult problem is specifying a virtual machine which can peacefully coexist with the desired target systems.

In some respects, this approach makes sense for any software development project. The identification of clear-cut interfaces is a standard structured programming technique, which (in theory at least) reduces software maintenance costs. The only controversy might be over the particular choice of structure (i.e., the virtual machine). In general, whenever organizational software is likely to outlive its hardware, the virtual operating system approach warrants consideration. This is because of the high redevelopment costs.

4. One Realization of a Virtual Operating System

To test the approach, a uniform program development environment was installed on several distinct systems. A program development environment consists of resources which assist programmers in the development and maintenance of computer programs, such as text editors, programming language processors, and file systems. The types of system resources with which such a virtual machine is concerned (files, directories, processes, and the user environment) require a general-purpose operating system interface.

Since the primary goal was to achieve some practical results, the system was to be modeled after an existing real operating system. The major criteria for the selection of this real system were the popularity of it within its user community and the estimated relevance of it to the programming needs within the organization. After an extensive survey of existing systems, the Unix[1] operating system [3] appeared to be a good candidate for emulation.

The actual virtual machine implementation permits the manipulation of files, directories, processes, and the user environment. The complete list of primitives implemented are given in Appendix A. Most of the file manipulation primitives were adopted from Kernighan and Plauger's *Software Tools* [8], since these primitives already provided a virtual machine consistent with a subset of the Unix system. This virtual machine could be used to implement most of the program development environments currently available. In particular, it permitted the implementation of many of the text manipulation utilities of the Unix system, as well as a command line interpreter similar to the Unix shell [4].

The primary requirements in the selection of a programming language for the virtual operating system were that the resulting code be portable to a wide range of machine architectures and that there be a substantial body of existing code upon which to base the system. The language chosen was RATFOR (rational Fortran) [7], a Fortran preprocessor which includes a reasonable set of flow-control structures (**if-else, while, for,** and **repeat-until**). This choice meets the two requirements, since ANSI-66 Fortran [1] compilers are available for use in most vendor environments and since the source code for the utilities in [8] is available in machine readable form. (These were implementations of many of the Unix text-processing utilities.) An extra plus was that RATFOR represented a reasonable way to encourage structured programming, since Fortran was already the predominant programming language.

Although the implementation of the utility programs was greatly aided by the availability of the source code [8], a fair amount of effort was necessary to increase the appeal of the system within a user community. In particular, all of the original utilities were substantially enhanced, and new ones were written as their needs were perceived. To complete the implementation of the virtual operating system, the command line interpreter was written, again emulating that of Unix [4]. On-line documentation of the system was provided [5], and a guide for installing the package on new systems was written [9]. In all cases, the system was offered in parallel with the existing environment, thereby allowing users to experiment with the virtual operating system without giving up the familiar, vendor-supplied environment. A complete list of the utilities in the system is presented in Appendix B. A description of the command interpreter is provided in Appendix C.

To encourage experimentation and alleviate user frustration, the source code for the system was made available to all interested parties, implicitly designating the universe of users as the system-programming group. It was felt that the resulting variation would complicate maintenance initially, but that the eventual benefits might outweigh the disadvantages.

5. Experience

5.1 Achieving Functional Equivalence of Operating Systems

The virtual operating system was implemented on the systems listed in Table I. A virtual operating system based upon a restricted set of the primitives of Appendix A was implemented on a much wider variety of machine architectures, as shown in Appendix D. The implementations listed in Table I indicate that these

[1] Unix is a trademark of Bell Laboratories.

Table I.

Vendor	Machine	Operating system	Person-months
CDC	6000	BKY	4
DEC	11/70	IAS	2
DEC	11/780	VMS	1
DEC	11/34	RSX-11M	3
DEC	PDP-10	TENEX	2
Modcomp IV		MAX4	5

vendor-supplied operating systems supply most of the system calls necessary to implement this virtual machine.

Complete uniformity across the different vendors may require modification of one or more of the host operating systems. This usually invalidates vendor-software maintenance contracts. Fortunately, a knowledgeable system programmer can often solve the problem through creative primitive implementation. But regardless of the manner in which the virtual machine is implemented on existing machines, the mappability of the virtual machine may be used as a criterion for selecting prospective vendors.

The following is an example of one apparent nonuniformity. Most multiprogramming operating systems supply a central portion of the executive which handles the communication with user terminals (the "terminal handler"). Certain keys on the terminal keyboard have special meaning to the terminal handler—e.g., erase previous character, interrupt process, and suspend terminal output. Even though there is a standard [2] for the interpretation of the character codes generated by the terminals, most systems apply their own semantics to the nonprinting ones, with the result that the keyboard interfaces to different systems are extremely nonuniform. To complicate the situation, these semantics are usually not under the control of the user. User mobility in this situation is thus severely hindered.

One solution to this problem is to modify the terminal handler for each system to present a common keyboard interface on all systems, with the side-effect of invalidating software maintenance contracts. Fortunately, most systems also provide the capability of transmitting and receiving characters with no interpretation by the terminal handler ("raw terminal I/O"). If the virtual machine I/O primitives transfer raw I/O to and from terminals, then a common set of semantics may be applied to the character codes on all systems, thus creating a uniform keyboard interface. Systems which do not allow user-applied semantics to the character codes or do not permit raw terminal I/O can be avoided by organizations wishing to preserve this common keyboard interface.

This is not the only example of the difficulties encountered in such an endeavor, but it is illustrative in the sense that it indicates most problems can be solved without resorting to modification of the vendor software.

In conclusion, the functional equivalence of vendor operating systems is strongly dependent upon the virtual machine specified. In the case of the machine outlined in Section 4, the virtual operating system primitives are implementable over a wide range of machine architectures without modification to the host operating system. A more general conclusion is that if the virtual machine specification accurately represents the needs of a particular organization, the implementability of the virtual machine is the major criterion in the selection of a new computer system.

5.2 Estimating Costs

There are two types of costs incurred when using a virtual operating system approach:

(1) The costs of writing the utilities: This is a one-time cost, since these utilities are independent of any real operating system. The program development costs for the utilities will be similar to those for any other software system designed for a specific machine, since the virtual operating system utilities are designed for the virtual machine.

(2) The costs of implementing the virtual machine: These are incurred once for each different host operating system within the organization. It is important to note that this is the only cost in moving all personnel and software to the new computing environment.

It has been estimated[2] that eight to ten person-months of effort were required to implement the original utilities in [8]. In addition, six to eight person-months were spent enhancing these original utilities. The largest single investment in new code was writing the command line interpreter, which required four person-months. In all, approximately two person-years have been invested in the implementation of the utilities of Appendix B.

The costs incurred in the implementation of the virtual machine on several systems are given in Table I. It is notable that the average time necessary to port the entire system was approximately four person-months. The dominance of Digital Equipment Corporation systems should not be interpreted as a lack of rigorous testing of the concept, since the operating systems on these machines are quite different.

In cases such as this, in which the effort required to implement the virtual machine is small, an initial attempt at implementation can be made as part of the evaluation of new systems. The decision to purchase can then be based upon whether or not the virtual machine

[2] Brian Kernighan, private communication: "... Probably 8–10 person months, but we were writing the book too. (That's 4-5 months for two people.)"

is implementable on the given system. Movement of personnel and software can be essentially instantaneous.

5.3 Optimizing Machine Efficiency

The issue of machine efficiency (the ability to minimize the demands of the software upon scarce hardware and software resources) is addressed through the design and implementation of the virtual machine. The virtual machine selected indicates those resources which the utilities can manipulate and outlines any possible bottlenecks in the utilization of those resources.

The utilities of the virtual operating system described here are primarily oriented toward text processing (source code generation, documentation, inter-user communication, etc.) These types of utilities are characteristically bounded by input/output rates [8]. Since the input/output capabilities are isolated in the virtual machine, the effect of this particular problem can be reduced through efficient implementation of the I/O primitives.

The effect of the programming language on efficiency should also be studied. Snow [11] has reported on the automatic translation of RATFOR to BCPL [10], which resulted in a substantial reduction in memory requirements and enhanced execution speeds. Preliminary investigations at the Lawrence Berkeley Laboratory (LBL) have indicated that a 50 percent reduction in object code size and a 30 percent improvement in CPU utilization are attainable on a VAX-11/780 running the VMS operating system by automatically translating RATFOR to BLISS [12]. Table II summarizes code size and execution time for various language translation alternatives. The example is "scopy," a frequently used string copy routine.

As a rule, it is necessary to anticipate bottlenecks in resource utilization during the design phase of the virtual machine. If manipulation of these resources is restricted to the virtual machine, efficiency can be achieved through optimization of the primitives alone. All utilities accessing these resources receive the benefits of such optimization automatically.

5.4 Proliferation of Variants

When a collection of diverse systems share a common user interface, a uniform environment is said to exist. It is this uniform environment which makes the virtual operating system approach appealing. The existence of variants destroys this uniformity. The distribution of source code to users invites the proliferation of variants. The traditional method of controlling this is to restrict development to a small group of experts. However, this method tends to produce user frustration and inhibit system growth, often resulting in mediocrity.

Although such variants are bothersome and undesirable, they are necessary for growth, and are analogous, say, to genetic variations in a biological population. As conditions change, software that can be adapted to changing requirements will survive. The abstract virtual machine and high-level language used in a virtual operating system enable the software to be adapted to changing conditions.

When software is used by many organizations, a user group may perform the control functions necessary to limit variation. To test this particular scenario, a user group was organized. Current activities of the group include the establishment of a centralized distribution facility, distribution of a newsletter, organization of active special interest groups on various topics, and sponsorship of biannual meetings. Standards for the various utilities are expected to result from such activities. In this manner, a benign form of control over the variation of the code is exercised.

6. Conclusions

Significant progress toward disentangling computing environments from their underlying operating system has been made. Using the virtual operating system approach, uniformity can be achieved at the three principal levels of user interface—the virtual machine, the system utilities, and the command language.

For at least one realization of the virtual machine interface, the functional equivalence of vendor operating systems has been established. Complete uniformity of environment is achievable without disturbing vendor software.

Although the effort to install a virtual operating system is large when compared to the effort required when moving a single program, it is small when compared to the cost of moving an organization's software. Moreover, when personnel retraining costs are considered, installation costs are insignificant. The approach permits accurate estimation of the cost of moving to a new system. The cost of moving people is zero, and the cost of software is equal to the cost of implementing the virtual machine.

The question of machine efficiency can also be addressed. By anticipating bottlenecks in resource utilization, critical functions can be isolated and solutions incorporated in the architecture of the virtual machine. This permits the benefits to be shared by all software.

The proliferation of variants

Table II.

	Code size	Speed
Hand-coded assembly language	1.0	1.0
BLISS—simulated automatic translation	1.0	4.6
Fortran—hand-coded	3.0	4.6
RATFOR	3.0	6.0

brought on by wide distribution of source code does not appear to be a serious problem. It has been our experience that the formation of a user group helps standardize both utilities and the virtual machine.

Acknowledgments

The authors gratefully acknowledge the cooperation of B. Kernighan of Bell Labs, the Addison-Wesley Publishing Company, and the many individuals who implemented the package on other systems. A project of this magnitude necessarily involves many persons from numerous sites. The following provided especially helpful suggestions and comments: D. Austin, M. Bronson, and B. Upshaw of LBL, B. Calland of NOSC, D. Comer of Purdue, P. Enslow of Georgia Tech., D. Hanson of the University of Arizona, T. Layman of IAC, D. Martin of Hughes Aircraft, R. Munn of the University of Maryland, C. Petersen of ORINCON, and J. Pool of DOE Headquarters.

References

1. *American National Standard FORTRAN.* ANS X3.9–1966, Amer. Nat. Standards Inst., N.Y., 1966. Contains the official description of the programming language Fortran 66.

2. *American Standard Code for Information Interchange.* ANS X3.4–1977, Amer. Nat. Standards Inst., N.Y., 1977. Contains the official description of the data alphabet called ASCII.

3. *The Bell Syst. Tech. J. 57*, 6 (July–August 1978). Perhaps the best single source of Unix literature. The entire issue is devoted to the Unix time-sharing operating system.

4. Bourne S.R. The Unix shell. *The Bell Syst. Tech. J. 57*, 6 (July–August 1978), 1971–1990. Describes the official Unix command language.

5. Hall, D., Scherrer, D., and Sventek, J. The software tools programmers manual. Internal Rep. LBID 097, LBL, University of Calif., Berkeley, Calif., 1978. A manual for the program development environment described in this report. Describes the virtual machine, the utilities, and the command language in detail.

6. Brinch Hansen, P. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, N.J., 1973. Designed for readers with a background in programming and a knowledge of elementary calculus and probability theory, focuses on general concepts illustrated with algorithms, techniques, and performance figures from actual systems.

7. Kernighan, B.W. RATFOR—a preprocessor for a rational FORTRAN. *Software—Practice and Experience 5*, 4 Oct.–Dec. (1975), 395–406. Discusses design criteria for a Fortran preprocessor, the RATFOR language and its implementation, and user experience.

8. Kernighan, B., and Plauger, P. *Software Tools.* Addison-Wesley Pub. Co., ISBN O-201-03669-X, Reading, Mass., 1976. Presents the principles of good programming practice in the context of actual working programs. The code is available in machine-readable form as a supplement to the text.

9. Scherrer, D. COOKBOOK, instructions for implementing the LBL software tools package. Internal Rep. LBID 098, LBL, University of Calif., Berkeley, Calif., 1978. Provides guidelines for installing the software tools program development environment on new systems.

10. Richards, M. The portability of the BCPL compiler. *Software—Practice and Experience 1*, 2 (April–June 1971), 135–146. Describes a method for porting a BCPL compiler which includes the specification of OCODE, a language used as an interface between the machine-independent and machine-dependent parts of the compiler.

11. Snow, C.R. The software tools project. *Software—Practice and Experience 8*, 5 Sept.–Oct. (1978), 585–599. Describes an implementation project on a Burroughs B1700 computer using an automatic code translation technique.

12. Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: A language for systems programming. *Comm. ACM 14*, 12 (Dec. 1971), 780–790. Describes BLISS, a language designed to be especially suitable for use in writing production software systems for DEC machines.

Appendix A. Virtual Machine Primitives

The following summarizes the primitive functions of the virtual machine chosen to test the virtual operating system technique.

File Access

open	open a file for reading, writing, or both
create	create a new file (or overwrite an existing one)
close	close (detach) a file
remove	remove a file from the file system
tty	determine if file is a teletype/CRT device
gettyp	determine if file is character or binary

I/O

getch	read character from file
putch	write character to file
seek	move read/write pointer
markl	pick up position in file
readf	read "n" bytes from file
writef	write "n" bytes to file
flush	force flushing of I/O buffer

Process Control

spawn	execute subtask
pstat	determine status of process
kill	kill process
resume	resume process after a suspend
suspnd	suspend process

Directory Manipulation

opendr	open directory for reading
closdr	close directory
gdrprm	get next file name from directory
gdraux	get auxiliary file information from directory
mkpath	generate full Unix pathname from local file name
mklocl	generate local file specification from pathname
cwdir	change current directory
gwdir	get current working directory name
mkdir	create a directory
rmdir	delete a directory
mvdir	move (rename) directory

Miscellaneous

getarg	get command line arguments
delarg	delete command line argument "n"
initr4	initialize all standard I/O and common blocks
endr4	close all open files and terminate program
date	get current date and time
mailid	get name of current user and home directory

Quasi Primitives

Many of the following were defined as primitives in the original Kernighan-Plauger package. However, since it is possible to implement these in terms of previously defined primitives, or, in one case, to adjust the RATFOR preprocessor to handle the problem, it was decided to move these functions to the portable category. Nevertheless, optimization is usually advisable for increased efficiency or capability.

prompt	putlin with carriage return/line-feed suppressed
getlin	read next line from file
putlin	write a line to file
remark	print single-line message
scratf	generate unique (scratch) file name
amove	move (rename) file1 to file2

Appendix B. Utilities

The following summarizes the utility functions which constitute one portion of the program development environment. These emulate many of the utilities found in the Unix operating system.

ar	archive file maintainer
cat	concatenate and print text files
ccnt	character count
ch	make changes in text files
cmp	compare two files
comm	print lines common to two files
cpress	compress input files

crt	copy files to terminal
crypt	crypt and decrypt standard input
cwd	change working directory
date	print date and time
detab	convert tabs to spaces
echo	print command line arguments
ed	text editor
entab	convert spaces to tabs and spaces
expand	uncompress input files
find	search a file for a pattern
form	generate form letter
help	list on-line documentation
incl	expand included files
kill	kill process
kwic	make key word in context index
lcnt	line count
ls	list contents of directory
macro	process macro definitions
mail	send or receive mail
man	run off section of user's manual
mkdir	create a directory
mv	move (rename) a file
mvdir	move (rename) a directory
os	(overstrike) convert backspaces into multiple lines
postmn	see if user has mail
pstat	check process status
pwd	print working directory
rat4	RATFOR preprocessor
resolve	identify mail users
resume	resume suspended process
rm	remove files
roff	format text
rmdir	remove directory
sh	shell (command line interpreter)
sort	sort and/or merge text files
spell	find spelling errors
split	split a file into pieces
suspnd	suspend running process
tee	copy input to standard output and named files
tr	character transliteration
uniq	strip adjacent repeated lines from a file
unrot	unrotate lines rotated by kwic
wcnt	(character) word count

Appendix C. Command Language

The *shell* is a command interpreter: It provides a user interface to the process-related facilities of the virtual operating system. It executes commands that are read either from a terminal or from a file.

Commands

Simple commands are written as sequences of "words" separated by blanks. The first word is the name of the command to be executed, and any remaining words are passed as arguments to the invoked command. The command name actually specifies a file which should be brought into memory and executed. If the file cannot be found in the current directory (or through its pathname), the shell searches one or more specific directories of commands intended to be available to shell users in general.

Standard I/O

The utilities of the virtual operating system have three standard files associated with them: standard input, standard output, and standard error output. All three are initially assigned to the user's terminal, but may be redirected to a disk file for the duration of the command by preceding the file name argument with special characters:

"<name" causes the file "name" to be used as the standard input file of the associated command.

">name" causes file "name" to be used as the standard output (">>name" appends it to the end of the file).

"?name" causes the file "name" to be used as the standard error output ("??name" appends the error message to the end of the file).

Most utilities also can read their input from a series of files simply by having the files listed as arguments to the command.

Filters and Pipes

The output from one command may be directed to the input of another. A sequence of commands separated by vertical bars (|) or carets ("^") causes the shell to arrange delivery of the standard output of each command to the standard input of the next command, in sequence. For example, the command line:

```
tr <name A-Z a-z | sort | uniq
```

translates all the upper case characters in file "name" to lower case, sorts them, and then strips out multiple occurrences of lines.

The vertical bar is called a "pipe." Programs such as tr, sort, and uniq, which copy standard input to standard output (making some changes along the way), are called filters.

Command Separators and Groupings

Commands need not be on different lines; they may be separated by semicolons.

The shell also allows commands to be grouped together by using parentheses so the group can then be used as a filter. For example,

```
(find <file1 this; find <file2 that) | sort
```

locates all lines containing "this" in file1, plus all lines containing "that" in file2, and sorts them together.

Multitasking

On many systems the shell also allows processes to be executed in the background. That is, the shell will not wait for the command to finish executing before prompting again. Any command may be run in background by following it with the operator "&".

Script Files

The shell may be used to read and execute commands contained in a file. Such a file is called a "script file." It can be used wherever a regular command can be issued. Arguments supplied with the call are referenced within the shell procedure by using the positional parameters $1, $2, etc.

Script files sometimes require in-line data to be available. A special input redirection notation "≪" is used to achieve this. For example, the editor normally takes its commands from the standard input. However, within a script file commands could be embedded as:

```
ed file ≪!
... editing requests
!
```

The lines between ≪! and ! are called a "here" document: they are read by the shell and made available as the standard input. The character "!" is arbitrary; the document is terminated by using a line which consists of the character that followed the ≪.

Shell Flags

The shell accepts several special arguments when it is invoked, causing it to print each line of a script file as it is read and/or executed, or to suppress execution of the command entirely, or to read the remaining arguments and execute them as a shell command.

Appendix D. Machines and Systems

The following summarizes the machines and systems used by members of the software tools user group. Most support at least the RATFOR preprocessor and the I/O primitives.

Burroughs B1700	local
CDC 1784	local
CDC 6000s, Cybers	KRONOS, UT-2D, local, DUAL-MACE, SCOPE3, NOS
CDC 7600	LTSS, SCOPE II, local
CDC MP-32	MPX/OS
Cray	CPSS
DataGeneral Eclipse (C & S series)	AOS, RDOS
DataGeneral Nova	RDOS
DataGeneral MP-100	MP/OS
ROLM 1602	RDOS
GEC 4070	OS 4000
Honeywell 6000S	GCOS-3
Honeywell Level 6	MOD 6 OS
Multics	Multics

ACOS 700	GCOS
AN/UYK-20	Level 2
HP 1000, 3000	RTE-IVB, MPE-III
HP 21MX	RTE III, RTE IV
IBM S/360, S/370, 303x	OS/MVT, VM/CMS, MVS, TSO, Wilbur
IBM 1130	DM2
FACOM M-200, M-190	OS IV/F4
HITAC 8700, 8800	OS7
M170	VOS3
Intel 8080	ISIS
Intel 8086	UCSD Pascal
Interdata 70	DOS
Interdata 8/32	OS/32MT
Modcomp	MAX
PDP 10	TOPS10, TYMCOM-X, TENEX
PDP 11s	RSX-11M, RSX-11S, RSX-11D, IAS, RT-11, RSTS, Unix, DOS, S
PDP 15	XVM/RSX
PDP 20	TOPS20
VAX	VMS
LSI 11	UCSD Pascal, RT-11, DOS-2
Prime	PRIMOS
SEL 32/77	MPX
SIEMANS 4004	TST
TELEFUNKEN TR440	BS19
Univac 1100	EXEC 8
Univac 90/70	VS/9
Xerox Sigma	RBM, CP-V
Zilog Z80	CP/M, Oasis