Data stream linkage and the UNIX system

To the Editor:

In J. P. Morrison's excellent paper "Data Stream Linkage Mechanism,"[1] it is surprising that he makes no reference to a widely used system that embodies many of the ideas he discusses—that is, the UNIX time-sharing system.[2,3] This system, which runs on a variety of computers, allows users to quickly and easily connect programs together by typing simple commands. As Morrison predicts, this makes programming easier and faster, and it also allows existing programs to be hooked together in new ways to solve new problems.

In the UNIX environment, every program that is run has a standard output file and a standard input file. Unless otherwise specified, these files are directed to the user's terminal so that the output from a command is normally presented to the user, and input is read from the terminal keyboard. The input and output can be diverted to disk files, devices such as printers and tapes, or, most important, to other programs. Some examples may clarify this: The command *ls* lists the names of the user's files at the terminal, and

Forum

ls >file.names

puts the names into the file *file.names*. (The character > means to direct output to a file.) There is a formatting program called *pr* that adds titles to its input file, optionally puts the file into a multicolumn format, and puts the result on its standard output. The command line

ls | pr −2 −h "My files"

prints the file names in two columns with a header, and

ls | pr −2 −h "My files" >names.out

puts the same text into the file *names.out*.

A program that performs such a stream operation is called a *filter*. Given the existence of a filter *lpr*, which sends its input to the high-speed printer spooler, then

ls | pr −2 −h "My files" | lpr

prints the names in two columns on the high-speed printer.

Programs are connected with special *pipe* files, which have the property of passing data from a writer to a reader. They can be created by any program when they are needed. The implementation is such that it is irrelevant whether a program is reading from or writing to a terminal, physical device, disk file, or pipe. The same READ and WRITE calls apply equally to each. Programs written in any combination of languages can be hooked together without special effort, since they all use the same system calls to do input and output. The pipe files are created by the command interpreter when the command line is typed, so that no special configuration step or precompilation is necessary. It is possible to have more complex arrangements of pipes than the simple linear arrangements shown here, but they seem to be much less generally useful. Similarly, it is possible to pass arbitrary binary data though a pipe, but usually it is most convenient to pass text.

Several interesting things have resulted from this view of programming. Rather than writing large programs with many options, people have tended to write simple, straightforward programs, and connect several together to achieve complicated effects. For example, the *ls* program mentioned above only lists file information. On most systems, it would need options to route the listing to the printer, control the listing format, etc., but in the UNIX system, these operations are performed by separate programs.

As a more complex example, there is a word processing program called *troff* which formats text for a typesetting machine (it is somewhat like SCRIPT, which is available on IBM systems). At one point some people wanted to add a facility to make the typesetting of mathematical text, including special symbols and equations, easier. Rather than modifying *troff*, they wrote a preprocessor called *eqn* which translates a convenient equation input language into the typesetter control codes that *troff* requires, but leaves normal text alone. Then a user enters

```
eqn <input.file | troff
```

and the file *input.file* is processed by *eqn* to do the equation handling, and then by *troff* to do the rest of the typesetting. Several other *troff* preprocessors have been written, and they can be strung together to do whatever processing is desired.

At another point, there was a need to detect spelling errors in documents. A dictionary of commonly used words was available, along with several utility programs, a transliteration filter, a general purpose sorting program, a filter to remove duplicated lines from a file, and a utility that compares two files and reports lines found in one but not in the other. These elements were combined into a pipeline that transliterates all strings of nonalphabetic characters into carriage returns (to put one word per line), sorts the

words, and removes duplicate occurrences of words, then prints words that occur in the document but not in the dictionary. Compare the few moments required in constructing this command line to the effort that would be needed to write the equivalent program in a conventional programming language.

For further details and many other examples, the reader is directed to Reference 3.

Experience with the UNIX system has shown that the pipeline approach to programming can drastically reduce the effort required to program new applications. Furthermore, a tool-oriented style of programming develops.[4] Each program is seen not as an end in itself, but rather as a tool, which, in combination with other tools (be they programs, data processing equipment, or whatever) can be used to get a job done. A well written program is like a screwdriver in that it is designed for only one job and does that job well, but inevitably it is used in many unanticipated ways to solve problems far removed from its originally intended application.

One certainly would hope that this successful experience will influence future computing systems and make the job of data processing easier.

CITED REFERENCES

1. J. P. Morrison, "Data Stream Linkage Mechanism," *IBM Systems Journal* **17**, No. 4, 383-408 (1978).
2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**, No. 7, 365-375 (July 1974).
3. *Bell System Technical Journal* **57**, No. 6, Part 2 (July-August 1978).
4. B. W. Kernighan and P. J. Plaugher, *Software Tools*, Addison-Wesley, Reading, Massachusetts (1976).

John R. Levine
Data Processing Consultant
5532 Yale Station
New Haven, Connecticut 06520

Author's response

Mr. Levine is correct in observing that UNIX should have been mentioned in my paper. I assumed that its "pipeline" concept was simply a shorthand added to a conventional interactive command language, and that therefore it was not relevant to my topic. I now find, however, that the concept does have relevance.

Following the publication of my paper, J. R. Mashey of Bell Telephone Laboratories kindly sent me a number of documents that describe UNIX.[1,2,3] In addition, I have been in communication

with A. Springer of the IBM Cambridge Scientific Center, who has made some comparisons between UNIX and VM/370 CMS, the IBM system which UNIX most closely resembles.[4] Consequently I feel that I have a much better understanding of this interesting system, and I would like to make a few comments about the way UNIX, VM/370 CMS, and DSLM seem to relate to each other.

The basic building block of UNIX, as in DSLM, is the *process*, a program that runs asynchronously with other like programs, communicating with them by means of data streams. It is irrelevant where the input to a process comes from, or where its output goes (the terminal, files, or other processes may all be used), so processes can be configured into networks without internal modification (this is the basic requirement for what N. P. Edwards[5] calls a *configurable architecture*). UNIX has a command interpreter called the *shell*, which supports a powerful, concise command language, in which the user can specify how data flows between processes, which functions are to run asynchronously with others, and other relationships.

For instance, extending the example in Mr. Levine's letter, the UNIX command

as source >output & ls | pr −2 −h "Heading" | lpr

can be read, "Assemble *source*, directing the output to the file called *output*; meanwhile, generate a list of files, format it into a 'two-up' listing with a specific heading, and send the result to the high-speed printer." The ampersand indicates that the assembly, once started, can proceed in parallel with the other processes. The character > means that the *standard output* of the assembly, which normally would go to the terminal, is to go to a file called *output* (the character < would do the same for the *standard input*). The other three processes are linked by the *pipe* operator |, which indicates that the standard output of one process is to be connected to the standard input of another. Two or more processes connected by pipes form a *pipeline* in UNIX terminology. Thus the user actually has started four asynchronous processes just by entering one command line at his terminal.

A quotation from Kernighan and Mashey[3] could be applied to DSLM simply by reading *queue* for *pipe*: "Programs connected by a pipe run concurrently, with the system taking care of buffering and synchronization. The programs themselves are oblivious to the I/O redirection. . . . The syntax is again concise and natural; *pipes are readily taught to nonprogramming users*." (My italics.)

The article continues: "Although in principle the pipe notation could be merely a shorthand for the longer form with temporaries [temporary files to which input and output could be redirected using < and >], there are significant advantages in running the

processes concurrently, with hidden buffers instead of files serving as the data channels. . . ." The authors go on to list a number of the advantages.

Given that a process is independent of the source of its input and of the destination of its output, it is not surprising that a UNIX process has points of attachment (like DSLM's *ports*) that are identified by numbers called *file descriptors*, which are relative to the process itself. File descriptors 0, 1, and 2 are reserved, respectively, for the functions of *standard input*, *standard output* (the ones used by the *pipe* operator), and *diagnostic output*, while descriptors 3 and up are available for process-specific files. (Standard input and output are initially assigned to the user's terminal, but they may be directed to files by the shell symbols < and > for input and output, respectively, or to other processes by the *pipe* operator.)

A difference of orientation between DSLM and UNIX is that, in DSLM, a network normally is specified for an application during the design process, whereas in UNIX, network specification is controlled dynamically by the user. Several people have commented that DSLM could be enhanced by allowing dynamic network specification. Thus, network specification in UNIX is "bound later" than in DSLM, but it is conceptually similar.

UNIX has marked similarities to VM/370 CMS in that both systems support a command language in which the user constructs and executes lists of commands for frequently used functions that are more complex than a single command. In UNIX these command lists are called *shell procedures*, and in CMS, *EXEC's*. In both systems, a given command list can invoke other command lists. In this way, every user builds his own personal "tool kit" of useful functions. Many applications can be written entirely in the command language, and the performance often is good enough that no further programming is required. If improved performance is desired, the user can do some performance evaluation and pick the parts of the application that need to be enhanced.

One way in which CMS modules differ from UNIX processes is that a CMS function usually is not independent of the source or the destination of its input and output, unless independence was designed in from the start (apart from the *console stack* mechanism, which allows console commands to be obtained from a file). In UNIX, on the other hand, all files look like streams of data bytes, and to programs it is irrelevant where these streams are coming from, or going. The same is true for a DSLM process, except that the streams consist of records rather than bytes.

A further difference is that a CMS module cannot run overlapped with other modules in the same virtual machine. Users often find

that, when waiting for completion of a long-running job, such as an assembly, they would like to continue entering commands but cannot do so unless they use two virtual machines. In UNIX, on the other hand, a single character suffices to specify that a process is to run asynchronously with other programs.

In all three systems, applications tend to be implemented not as large, monolithic systems, which would require long coding stages before any results could be observed, but as structures of many small, function-oriented modules, which can be connected together, and tested individually or in networks of gradually increasing size.

I am encouraged by the fact that UNIX has been well received by a variety of users in a variety of environments, especially in universities (as of mid-1978, there were almost 1000 systems in operation around the world[3]). This acceptance suggests that data-linked asynchronous processes, far from being complex and esoteric devices, in fact constitute a powerful and natural way of instructing computers to do what we want them to do. Comparison of the three systems suggests that combining their essential features would result in a powerful, natural, and easy-to-use man-machine interface for both data processing professionals and other users.

CITED REFERENCES

1. *The Bell System Technical Journal* **57,** No. 6, Part 2 (July-August 1978).
2. J. R. Mashey, "PWB/UNIX Shell Tutorial," Bell Telephone Laboratories, Murray Hill, New Jersey 07974 (September 1977).
3. B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *Software—Practice and Experience* **9,** 1-15 (1979).
4. For detailed information on VM/370 and CMS, see *IBM Systems Journal* **18,** No. 1 (1979).
5. N. P. Edwards, *On the architectural requirements of an engineered system*, Research Report RC 6688, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (August 1977). (ITIRC AAA 77A004397.)

J. Paul Morrison
IBM Canada, Ltd.
Finance Industry Support and Development
Toronto, Ontario M5K 1B1

Reprint Order No. G321-5105.