The Evolution of the Unix Time-sharing System*

Dennis M. Ritchie

Bell Labs, Lucent Technologies
Murray Hill, New Jersey 07974

ABSTRACT

This paper presents a brief history of the early development of the Unix operating system. It concentrates on the evolution of the file system, the process-control mechanism, and the idea of pipelined commands. Some attention is paid to social conditions during the development of the system.

Introduction

During the past few years, the Unix operating system has come into wide use, so wide that its very name has become a trademark of Bell Laboratories. Its important characteristics have become known to many people. It has suffered much rewriting and tinkering since the first publication describing it in 1974 [1], but few fundamental changes. However, Unix was born in 1969 not 1974, and the account of its development makes a little-known and perhaps instructive story. This paper presents a technical and social history of the evolution of the system.

Origins

For computer science at Bell Laboratories, the period 1968-1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the Multics project. To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier. For much of this time, the Murray Hill Computer Center was also running a costly GE 645 machine that inadequately simulated the GE 635. Another shake-up that occurred during this period was the organizational separation of computing services and computing research.

From the point of view of the group that was to be most involved in the beginnings of Unix (K. Thompson, Ritchie, M. D. McIlroy, J. F. Ossanna), the decline and fall of Multics had a directly felt effect. We were among the last Bell Laboratories holdouts actually working on Multics, so we still felt some sort of stake in its success. More important, the convenient interactive computing service that Multics had promised to the entire community was in fact available to our limited group, at first under the CTSS system used to develop Multics, and later under Multics itself. Even though Multics could not then support many users, it could support us, albeit at exorbitant cost. We didn't want to lose the pleasant niche we occupied, because no similar ones were available; even the time-sharing service that would later be offered under GE's operating system did not exist. What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.

Thus, during 1969, we began trying to find an alternative to Multics. The search took several forms.

---

*This paper was first presented at the Language Design and Programming Methodology conference at Sydney, Australia, September 1979. The conference proceedings were published as *Lecture Notes in Computer Science #79: Language Design and Programming Methodology,* Springer-Verlag, 1980. This rendition is based on a reprinted version appearing in AT&T Bell Laboratories Technical Journal **63** No. 6 Part 2, October 1984, pp. 1577-93.

Throughout 1969 we (mainly Ossanna, Thompson, Ritchie) lobbied intensively for the purchase of a medium-scale machine for which we promised to write an operating system; the machines we suggested were the DEC PDP-10 and the SDS (later Xerox) Sigma 7. The effort was frustrating, because our proposals were never clearly and finally turned down, but yet were certainly never accepted. Several times it seemed we were very near success. The final blow to this effort came when we presented an exquisitely complicated proposal, designed to minimize financial outlay, that involved some outright purchase, some third-party lease, and a plan to turn in a DEC KA-10 processor on the soon-to-be-announced and more capable KI-10. The proposal was rejected, and rumor soon had it that W. O. Baker (then vice-president of Research) had reacted to it with the comment ‘Bell Laboratories just doesn’t do business this way!’

Actually, it is perfectly obvious in retrospect (and should have been at the time) that we were asking the Labs to spend too much money on too few people with too vague a plan. Moreover, I am quite sure that at that time operating systems were not, for our management, an attractive area in which to support work. They were in the process of extricating themselves not only from an operating system development effort that had failed, but from running the local Computation Center. Thus it may have seemed that buying a machine such as we suggested might lead on the one hand to yet another Multics, or on the other, if we produced something useful, to yet another Comp Center for them to be responsible for.

Besides the financial agitations that took place in 1969, there was technical work also. Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of Unix. Most of the design was Thompson’s, as was the impulse to think about file systems at all, but I believe I contributed the idea of device files. Thompson’s itch for creation of an operating system took several forms during this period; he also wrote (on Multics) a fairly detailed simulation of the performance of the proposed file system design and of paging behavior of programs. In addition, he started work on a new operating system for the GE-645, going as far as writing an assembler for the machine and a rudimentary operating system kernel whose greatest achievement, so far as I remember, was to type a greeting message. The complexity of the machine was such that a mere message was already a fairly notable accomplishment, but when it became clear that the lifetime of the 645 at the Labs was measured in months, the work was dropped.

Also during 1969, Thompson developed the game of ‘Space Travel.’ First written on Multics, then transliterated into Fortran for GECOS (the operating system for the GE, later Honeywell, 635), it was nothing less than a simulation of the movement of the major bodies of the Solar System, with the player guiding a ship here and there, observing the scenery, and attempting to land on the various planets and moons. The GECOS version was unsatisfactory in two important respects: first, the display of the state of the game was jerky and hard to control because one had to type commands at it, and second, a game cost about $75 for CPU time on the big computer. It did not take long, therefore, for Thompson to find a little-used PDP-7 computer with an excellent display processor; the whole system was used as a Graphic-II terminal. He and I rewrote Space Travel to run on this machine. The undertaking was more ambitious than it might seem; because we disdained all existing software, we had to write a floating-point arithmetic package, the pointwise specification of the graphic characters for the display, and a debugging subsystem that continuously displayed the contents of typed-in locations in a corner of the screen. All this was written in assembly language for a cross-assembler that ran under GECOS and produced paper tapes to be carried to the PDP-7.

Space Travel, though it made a very attractive game, served mainly as an introduction to the clumsy technology of preparing programs for the PDP-7. Soon Thompson began implementing the paper file system (perhaps ‘chalk file system’ would be more accurate) that had been designed earlier. A file system without a way to exercise it is a sterile proposition, so he proceeded to flesh it out with the other requirements for a working operating system, in particular the notion of processes. Then came a small set of user-level utilities: the means to copy, print, delete, and edit files, and of course a simple command interpreter (shell). Up to this time all the programs were written using GECOS and files were transferred to the PDP-7 on paper tape; but once an assembler was completed the system was able to support itself. Although it was not until well into 1970 that Brian Kernighan suggested the name ‘Unix,’ in a somewhat treacherous pun on ‘Multics,’ the operating system we know today was born.

The PDP-7 Unix file system

Structurally, the file system of PDP-7 Unix was nearly identical to today's. It had

1) An i-list: a linear array of *i-nodes* each describing a file. An i-node contained less than it does now, but the essential information was the same: the protection mode of the file, its type and size, and the list of physical blocks holding the contents.

2) Directories: a special kind of file containing a sequence of names and the associated i-number.

3) Special files describing devices. The device specification was not contained explicitly in the i-node, but was instead encoded in the number: specific i-numbers corresponded to specific files.

The important file system calls were also present from the start. Read, write, open, creat (sic), close: with one very important exception, discussed below, they were similar to what one finds now. A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters. Another minor, occasionally annoying difference was the lack of erase and kill processing for terminals. Terminals, in effect, were always in raw mode. Only a few programs (notably the shell and the editor) bothered to implement erase-kill processing.

In spite of its considerable similarity to the current file system, the PDP-7 file system was in one way remarkably different: there were no path names, and each file-name argument to the system was a simple name (without '/') taken relative to the current directory. Links, in the usual Unix sense, did exist. Together with an elaborate set of conventions, they were the principal means by which the lack of path names became acceptable.

The *link* call took the form

```
link(dir, file, newname)
```

where *dir* was a directory file in the current directory, *file* an existing entry in that directory, and *newname* the name of the link, which was added to the current directory. Because *dir* needed to be in the current directory, it is evident that today's prohibition against links to directories was not enforced; the PDP-7 Unix file system had the shape of a general directed graph.

So that every user did not need to maintain a link to all directories of interest, there existed a directory called *dd* that contained entries for the directory of each user. Thus, to make a link to file *x* in directory *ken*, I might do

```
ln dd ken ken
ln ken x x
rm ken
```

This scheme rendered subdirectories sufficiently hard to use as to make them unused in practice. Another important barrier was that there was no way to create a directory while the system was running; all were made during recreation of the file system from paper tape, so that directories were in effect a nonrenewable resource.

The *dd* convention made the *chdir* command relatively convenient. It took multiple arguments, and switched the current directory to each named directory in turn. Thus

```
chdir dd ken
```

would move to directory *ken*. (Incidentally, *chdir* was spelled *ch*; why this was expanded when we went to the PDP-11 I don't remember.)

The most serious inconvenience of the implementation of the file system, aside from the lack of path names, was the difficulty of changing its configuration; as mentioned, directories and special files were both made only when the disk was recreated. Installation of a new device was very painful, because the code for devices was spread widely throughout the system; for example there were several loops that visited each device in turn. Not surprisingly, there was no notion of mounting a removable disk pack, because the machine had only a single fixed-head disk.

The operating system code that implemented this file system was a drastically simplified version of the present scheme. One important simplification followed from the fact that the system was not multiprogrammed; only one program was in memory at a time, and control was passed between processes only when an explicit swap took place. So, for example, there was an *iget* routine that made a named i-node available, but it left the i-node in a constant, static location rather than returning a pointer into a large table of active i-nodes. A precursor of the current buffering mechanism was present (with about 4 buffers) but there was essentially no overlap of disk I/O with computation. This was avoided not merely for simplicity. The disk attached to the PDP-7 was fast for its time; it transferred one 18-bit word every 2 microseconds. On the other hand, the PDP-7 itself had a memory cycle time of 1 microsecond, and most instructions took 2 cycles (one for the instruction itself, one for the operand). However, indirectly addressed instructions required 3 cycles, and indirection was quite common, because the machine had no index registers. Finally, the DMA controller was unable to access memory during an instruction. The upshot was that the disk would incur overrun errors if any indirectly-addressed instructions were executed while it was transferring. Thus control could not be returned to the user, nor in fact could general system code be executed, with the disk running. The interrupt routines for the clock and terminals, which needed to be runnable at all times, had to be coded in very strange fashion to avoid indirection.

Process control

By ‘process control,’ I mean the mechanisms by which processes are created and used; today the system calls *fork*, *exec*, *wait*, and *exit* implement these mechanisms. Unlike the file system, which existed in nearly its present form from the earliest days, the process control scheme underwent considerable mutation after PDP-7 Unix was already in use. (The introduction of path names in the PDP-11 system was certainly a considerable notational advance, but not a change in fundamental structure.)

Today, the way in which commands are executed by the shell can be summarized as follows:

1) The shell reads a command line from the terminal.
2) It creates a child process by *fork.*
3) The child process uses *exec* to call in the command from a file.
4) Meanwhile, the parent shell uses *wait* to wait for the child (command) process to terminate by calling *exit.*
5) The parent shell goes back to step 1).

Processes (independently executing entities) existed very early in PDP-7 Unix. There were in fact precisely two of them, one for each of the two terminals attached to the machine. There was no *fork*, *wait*, or *exec*. There was an *exit*, but its meaning was rather different, as will be seen. The main loop of the shell went as follows.

1) The shell closed all its open files, then opened the terminal special file for standard input and output (file descriptors 0 and 1).
2) It read a command line from the terminal.
3) It linked to the file specifying the command, opened the file, and removed the link. Then it copied a small bootstrap program to the top of memory and jumped to it; this bootstrap program read in the file over the shell code, then jumped to the first location of the command (in effect an *exec*).
4) The command did its work, then terminated by calling *exit*. The *exit* call caused the system to read in a fresh copy of the shell over the terminated command, then to jump to its start (and thus in effect to go to step 1).

The most interesting thing about this primitive implementation is the degree to which it anticipated themes developed more fully later. True, it could support neither background processes nor shell command files (let alone pipes and filters); but IO redirection (via ‘<’ and ‘>’) was soon there; it is discussed below. The implementation of redirection was quite straightforward; in step 3) above the shell just replaced its standard input or output with the appropriate file. Crucial to subsequent development was the implementation of the shell as a user-level program stored in a file, rather than a part of the operating system.

The structure of this process control scheme, with one process per terminal, is similar to that of many interactive systems, for example CTSS, Multics, Honeywell TSS, and IBM TSS and TSO. In general such systems require special mechanisms to implement useful facilities such as detached computations and command files; Unix at that stage didn't bother to supply the special mechanisms. It also exhibited some irritating, idiosyncratic problems. For example, a newly recreated shell had to close all its open files both to get rid of any open files left by the command just executed and to rescind previous IO redirection. Then it had to reopen the special file corresponding to its terminal, in order to read a new command line. There was no */dev* directory (because no path names); moreover, the shell could retain no memory across commands, because it was reexecuted afresh after each command. Thus a further file system convention was required: each directory had to contain an entry *tty* for a special file that referred to the terminal of the process that opened it. If by accident one changed into some directory that lacked this entry, the shell would loop hopelessly; about the only remedy was to reboot. (Sometimes the missing link could be made from the other terminal.)

Process control in its modern form was designed and implemented within a couple of days. It is astonishing how easily it fitted into the existing system; at the same time it is easy to see how some of the slightly unusual features of the design are present precisely because they represented small, easily-coded changes to what existed. A good example is the separation of the *fork* and *exec* functions. The most common model for the creation of new processes involves specifying a program for the process to execute; in Unix, a forked process continues to run the same program as its parent until it performs an explicit *exec*. The separation of the functions is certainly not unique to Unix, and in fact it was present in the Berkeley time-sharing system [2], which was well-known to Thompson. Still, it seems reasonable to suppose that it exists in Unix mainly because of the ease with which *fork* could be implemented without changing much else. The system already handled multiple (i.e. two) processes; there was a process table, and the processes were swapped between main memory and the disk. The initial implementation of *fork* required only

1) Expansion of the process table

2) Addition of a fork call that copied the current process to the disk swap area, using the already existing swap IO primitives, and made some adjustments to the process table.

In fact, the PDP-7's *fork* call required precisely 27 lines of assembly code. Of course, other changes in the operating system and user programs were required, and some of them were rather interesting and unexpected. But a combined *fork-exec* would have been considerably more complicated, if only because *exec* as such did not exist; its function was already performed, using explicit IO, by the shell.

The *exit* system call, which previously read in a new copy of the shell (actually a sort of automatic *exec* but without arguments), simplified considerably; in the new version a process only had to clean out its process table entry, and give up control.

Curiously, the primitives that became *wait* were considerably more general than the present scheme. A pair of primitives sent one-word messages between named processes:

```
smes(pid, message)
(pid, message) = rmes()
```

The target process of *smes* did not need to have any ancestral relationship with the receiver, although the system provided no explicit mechanism for communicating process IDs except that *fork* returned to each of the parent and child the ID of its relative. Messages were not queued; a sender delayed until the receiver read the message.

The message facility was used as follows: the parent shell, after creating a process to execute a command, sent a message to the new process by *smes*; when the command terminated (assuming it did not try to read any messages) the shell's blocked *smes* call returned an error indication that the target process did not exist. Thus the shell's *smes* became, in effect, the equivalent of *wait*.

A different protocol, which took advantage of more of the generality offered by messages, was used between the initialization program and the shells for each terminal. The initialization process, whose ID was understood to be 1, created a shell for each of the terminals, and then issued *rmes*; each shell, when it read the end of its input file, used *smes* to send a conventional 'I am terminating' message to the initialization process, which recreated a new shell process for that terminal.

I can recall no other use of messages. This explains why the facility was replaced by the *wait* call of the present system, which is less general, but more directly applicable to the desired purpose. Possibly relevant also is the evident bug in the mechanism: if a command process attempted to use messages to communicate with other processes, it would disrupt the shell's synchronization. The shell depended on sending a message that was never received; if a command executed *rmes*, it would receive the shell's phony message, and cause the shell to read another input line just as if the command had terminated. If a need for general messages had manifested itself, the bug would have been repaired.

At any rate, the new process control scheme instantly rendered some very valuable features trivial to implement; for example detached processes (with '&') and recursive use of the shell as a command. Most systems have to supply some sort of special 'batch job submission' facility and a special command interpreter for files distinct from the one used interactively.

Although the multiple-process idea slipped in very easily indeed, there were some aftereffects that weren't anticipated. The most memorable of these became evident soon after the new system came up and apparently worked. In the midst of our jubilation, it was discovered that the *chdir* (change current directory) command had stopped working. There was much reading of code and anxious introspection about how the addition of *fork* could have broken the *chdir* call. Finally the truth dawned: in the old system *chdir* was an ordinary command; it adjusted the current directory of the (unique) process attached to the terminal. Under the new system, the *chdir* command correctly changed the current directory of the process created to execute it, but this process promptly terminated and had no effect whatsoever on its parent shell! It was necessary to make *chdir* a special command, executed internally within the shell. It turns out that several command-like functions have the same property, for example *login*.

Another mismatch between the system as it had been and the new process control scheme took longer to become evident. Originally, the read/write pointer associated with each open file was stored within the process that opened the file. (This pointer indicates where in the file the next read or write will take place.) The problem with this organization became evident only when we tried to use command files. Suppose a simple command file contains

```
ls
who
```

and it is executed as follows:

```
sh comfile >output
```

The sequence of events was

1) The main shell creates a new process, which opens *outfile* to receive the standard output and executes the shell recursively.

2) The new shell creates another process to execute *ls*, which correctly writes on file *output* and then terminates.

3) Another process is created to execute the next command. However, the IO pointer for the output is copied from that of the shell, and it is still 0, because the shell has never written on its output, and IO pointers are associated with processes. The effect is that the output of *who* overwrites and destroys the output of the preceding *ls* command.

Solution of this problem required creation of a new system table to contain the IO pointers of open files independently of the process in which they were opened.

IO Redirection

The very convenient notation for IO redirection, using the '>' and '<' characters, was not present from the very beginning of the PDP-7 Unix system, but it did appear quite early. Like much else in Unix, it was inspired by an idea from Multics. Multics has a rather general IO redirection mechanism [3] embodying named IO streams that can be dynamically redirected to various devices, files, and even through special stream-processing modules. Even in the version of Multics we were familiar with a decade ago, there existed a command that switched subsequent output normally destined for the terminal to a file, and another command to reattach output to the terminal. Where under Unix one might say

```
ls >xx
```

to get a listing of the names of one's files in *xx*, on Multics the notation was

```
iocall attach user_output file xx
list
iocall attach user_output syn user_i/o
```

Even though this very clumsy sequence was used often during the Multics days, and would have been utterly straightforward to integrate into the Multics shell, the idea did not occur to us or anyone else at the time. I speculate that the reason it did not was the sheer size of the Multics project: the implementors of the IO system were at Bell Labs in Murray Hill, while the shell was done at MIT. We didn't consider making changes to the shell (it was *their* program); correspondingly, the keepers of the shell may not even have known of the usefulness, albeit clumsiness, of *iocall*. (The 1969 Multics manual [4] lists *iocall* as an 'author-maintained,' that is non-standard, command.) Because both the Unix IO system and its shell were under the exclusive control of Thompson, when the right idea finally surfaced, it was a matter of an hour or so to implement it.

The advent of the PDP-11

By the beginning of 1970, PDP-7 Unix was a going concern. Primitive by today's standards, it was still capable of providing a more congenial programming environment than its alternatives. Nevertheless, it was clear that the PDP-7, a machine we didn't even own, was already obsolete, and its successors in the same line offered little of interest. In early 1970 we proposed acquisition of a PDP-11, which had just been introduced by Digital. In some sense, this proposal was merely the latest in the series of attempts that had been made throughout the preceding year. It differed in two important ways. First, the amount of money (about $65,000) was an order of magnitude less than what we had previously asked; second, the charter sought was not merely to write some (unspecified) operating system, but instead to create a system specifically designed for editing and formatting text, what might today be called a 'word-processing system.' The impetus for the proposal came mainly from J. F. Ossanna, who was then and until the end of his life interested in text processing. If our early proposals were too vague, this one was perhaps too specific; at first it too met with disfavor. Before long, however, funds were obtained through the efforts of L. E. McMahon and an order for a PDP-11 was placed in May.

The processor arrived at the end of the summer, but the PDP-11 was so new a product that no disk was available until December. In the meantime, a rudimentary, core-only version of Unix was written using a cross-assembler on the PDP-7. Most of the time, the machine sat in a corner, enumerating all the closed Knight's tours on a 6×8 chess board   a three-month job.

The first PDP-11 system

Once the disk arrived, the system was quickly completed. In internal structure, the first version of Unix for the PDP-11 represented a relatively minor advance over the PDP-7 system; writing it was largely a matter of transliteration. For example, there was no multi-programming; only one user program was present in core at any moment. On the other hand, there were important changes in the interface to the user: the present directory structure, with full path names, was in place, along with the modern form of *exec* and *wait*, and conveniences like character-erase and line-kill processing for terminals. Perhaps the most interesting thing about the enterprise was its small size: there were 24K bytes of core memory (16K for the system, 8K for user programs), and a disk with 1K blocks (512K bytes). Files were limited to 64K bytes.

At the time of the placement of the order for the PDP-11, it had seemed natural, or perhaps expedient, to promise a system dedicated to word processing. During the protracted arrival of the hardware, the increasing usefulness of PDP-7 Unix made it appropriate to justify creating PDP-11 Unix as a development tool, to be used in writing the more special-purpose system. By the spring of 1971, it was generally agreed that no one had the slightest interest in scrapping Unix. Therefore, we transliterated the *roff* text formatter into PDP-11 assembler language, starting from the PDP-7 version that had been transliterated from McIlroy's BCPL version on Multics, which had in turn been inspired by J. Saltzer's *runoff* program on CTSS. In early summer, editor and formatter in hand, we felt prepared to fulfill our charter by offering to supply a text-processing service to the Patent department for preparing patent applications. At the time, they were evaluating a commercial system for this purpose; the main advantages we offered (besides the dubious one of taking part in an in-house experiment) were two in number: first, we supported Teletype's model 37 terminals, which, with an extended type-box, could print most of the math symbols they required; second, we quickly endowed *roff* with the ability to produce line-numbered pages, which the Patent Office required and which the other system could not handle.

During the last half of 1971, we supported three typists from the Patent department, who spent the day busily typing, editing, and formatting patent applications, and meanwhile tried to carry on our own work. Unix has a reputation for supplying interesting services on modest hardware, and this period may mark a high point in the benefit/equipment ratio; on a machine with no memory protection and a single .5 MB disk, every test of a new program required care and boldness, because it could easily crash the system, and every few hours' work by the typists meant pushing out more information onto DECtape, because of the very small disk.

The experiment was trying but successful. Not only did the Patent department adopt Unix, and thus become the first of many groups at the Laboratories to ratify our work, but we achieved sufficient credibility to convince our own management to acquire one of the first PDP 11/45 systems made. We have accumulated much hardware since then, and labored continuously on the software, but because most of the interesting work has already been published, (e.g. on the system itself [1, 5, 6, 7, 8, 9]) it seems unnecessary to repeat it here.

Pipes

One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the *pipe*, as used in a pipeline of commands. Of course, the fundamental idea was by no means new; the pipeline is merely a specific form of coroutine. Even the implementation was not unprecedented, although we didn't know it at the time; the 'communication files' of the Dartmouth Time-Sharing System [10] did very nearly what Unix pipes do, though they seem not to have been exploited so fully.

Pipes appeared in Unix in 1972, well after the PDP-11 version of the system was in operation, at the suggestion (or perhaps insistence) of M. D. McIlroy, a long-time advocate of the non-hierarchical control flow that characterizes coroutines. Some years before pipes were implemented, he suggested that commands should be thought of as binary operators, whose left and right operand specified the input and output files. Thus a 'copy' utility would be commanded by

```
inputfile copy outputfile
```

To make a pipeline, command operators could be stacked up. Thus, to sort *input,* paginate it neatly, and print the result off-line, one would write

```
input sort paginate offprint
```

In today's system, this would correspond to

```
sort input | pr | opr
```

The idea, explained one afternoon on a blackboard, intrigued us but failed to ignite any immediate action. There were several objections to the idea as put: the infix notation seemed too radical (we were too accustomed to typing 'cp x y' to copy *x* to *y*); and we were unable to see how to distinguish command parameters from the input or output files. Also, the one-input one-output model of command execution seemed too confining. What a failure of imagination!

Some time later, thanks to McIlroy's persistence, pipes were finally installed in the operating system (a relatively simple job), and a new notation was introduced. It used the same characters as for I/O redirection. For example, the pipeline above might have been written

```
sort input >pr>opr>
```

The idea is that following a '>' may be either a file, to specify redirection of output to that file, or a command into which the output of the preceding command is directed as input. The trailing '>' was needed in the example to specify that the (nonexistent) output of *opr* should be directed to the console; otherwise the command *opr* would not have been executed at all; instead a file *opr* would have been created.

The new facility was enthusiastically received, and the term ‘filter’ was soon coined. Many commands were changed to make them usable in pipelines. For example, no one had imagined that anyone would want the *sort* or *pr* utility to sort or print its standard input if given no explicit arguments.

Soon some problems with the notation became evident. Most annoying was a silly lexical problem: the string after ‘>’ was delimited by blanks, so, to give a parameter to *pr* in the example, one had to quote:

```
sort input >"pr   2">opr>
```

Second, in attempt to give generality, the pipe notation accepted ‘<’ as an input redirection in a way corresponding to ‘>’; this meant that the notation was not unique. One could also write, for example,

```
opr <pr<"sort input"<
```

or even

```
pr <"sort input"< >opr>
```

The pipe notation using ‘<’ and ‘>’ survived only a couple of months; it was replaced by the present one that uses a unique operator to separate components of a pipeline. Although the old notation had a certain charm and inner consistency, the new one is certainly superior. Of course, it too has limitations. It is unabashedly linear, though there are situations in which multiple redirected inputs and outputs are called for. For example, what is the best way to compare the outputs of two programs? What is the appropriate notation for invoking a program with two parallel output streams?

I mentioned above in the section on IO redirection that Multics provided a mechanism by which IO streams could be directed through processing modules on the way to (or from) the device or file serving as source or sink. Thus it might seem that stream-splicing in Multics was the direct precursor of Unix pipes, as Multics IO redirection certainly was for its Unix version. In fact I do not think this is true, or is true only in a weak sense. Not only were coroutines well-known already, but their embodiment as Multics spliceable IO modules required that the modules be specially coded in such a way that they could be used for no other purpose. The genius of the Unix pipeline is precisely that it is constructed from the very same commands used constantly in simplex fashion. The mental leap needed to see this possibility and to invent the notation is large indeed.

High-level languages

Every program for the original PDP-7 Unix system was written in assembly language, and bare assembly language it was for example, there were no macros. Moreover, there was no loader or link-editor, so every program had to be complete in itself. The first interesting language to appear was a version of McClure’s TMG [11] that was implemented by McIlroy. Soon after TMG became available, Thompson decided that we could not pretend to offer a real computing service without Fortran, so he sat down to write a Fortran in TMG. As I recall, the intent to handle Fortran lasted about a week. What he produced instead was a definition of and a compiler for the new language B [12]. B was much influenced by the BCPL language [13]; other influences were Thompson’s taste for spartan syntax, and the very small space into which the compiler had to fit. The compiler produced simple interpretive code; although it and the programs it produced were rather slow, it made life much more pleasant. Once interfaces to the regular system calls were made available, we began once again to enjoy the benefits of using a reasonable language to write what are usually called ‘systems programs:’ compilers, assemblers, and the like. (Although some might consider the PL/I we used under Multics unreasonable, it was much better than assembly language.) Among other programs, the PDP-7 B cross-compiler for the PDP-11 was written in B, and in the course of time, the B compiler for the PDP-7 itself was transliterated from TMG into B.

When the PDP-11 arrived, B was moved to it almost immediately. In fact, a version of the multi-precision ‘desk calculator’ program *dc* was one of the earliest programs to run on the PDP-11, well before the disk arrived. However, B did not take over instantly. Only passing thought was given to rewriting the operating system in B rather than assembler, and the same was true of most of the utilities. Even the assembler was rewritten in assembler. This approach was taken mainly because of the slowness of the interpretive code. Of smaller but still real importance was the mismatch of the word-oriented B language with the byte-addressed PDP-11.

Thus, in 1971, work began on what was to become the C language [14]. The story of the language developments from BCPL through B to C is told elsewhere [15], and need not be repeated here. Perhaps the most important watershed occurred during 1973, when the operating system kernel was rewritten in C. It was at this point that the system assumed its modern form; the most far-reaching change was the introduction of multi-programming. There were few externally-visible changes, but the internal structure of the system became much more rational and general. The success of this effort convinced us that C was useful as a nearly universal tool for systems programming, instead of just a toy for simple applications.

Today, the only important Unix program still written in assembler is the assembler itself; virtually all the utility programs are in C, and so are most of the applications programs, although there are sites with many in Fortran, Pascal, and Algol 68 as well. It seems certain that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages.

Conclusion

One of the comforting things about old memories is their tendency to take on a rosy glow. The programming environment provided by the early versions of Unix seems, when described here, to be extremely harsh and primitive. I am sure that if forced back to the PDP-7 I would find it intolerably limiting and lacking in conveniences. Nevertheless, it did not seem so at the time; the memory fixes on what was good and what lasted, and on the joy of helping to create the improvements that made life better. In ten years, I hope we can look back with the same mixed impression of progress combined with continuity.

Acknowledgements

I am grateful to S. P. Morgan, K. Thompson, and M. D. McIlroy for providing early documents and digging up recollections.

Because I am most interested in describing the evolution of ideas, this paper attributes ideas and work to individuals only where it seems most important. The reader will not, on the average, go far wrong if he reads each occurrence of 'we' with unclear antecedent as 'Thompson, with some assistance from me.'

References

1. D. M. Ritchie and K. Thompson, 'The Unix Time-sharing System, C. ACM **17** No. 7 (July 1974), pp 365-37.
2. L. P. Deutch and B. W. Lampson, 'SDS 930 Time-sharing System Preliminary Reference Manual,' Doc. 30.10.10, Project Genie, Univ. Cal. at Berkeley (April 1965).
3. R. J. Feiertag and E. I. Organick, 'The Multics input-output system,' Proc. Third Symposium on Operating Systems Principles, October 18-20, 1971, pp. 35-41.
4. *The Multiplexed Information and Computing Service: Programmers' Manual,* Mass. Inst. of Technology, Project MAC, Cambridge MA, (1969).
5. K. Thompson, 'Unix Implementation,' Bell System Tech J. **57** No. 6, (July-August 1978), pp. 1931-46.
6. S. C. Johnson and D. M. Ritchie, Portability of C Programs and the Unix System,' Bell System Tech J. **57** No. 6, (July-August 1978), pp. 2021-48.
7. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna. 'Document Preparation,' Bell Sys. Tech. J., **57** No. 6, pp. 2115-2135.
8. B. W. Kernighan and L. L. Cherry, 'A System for Typesetting Mathematics,' J. Comm. Assoc. Comp. Mach. **18,** pp. 151-157 (March 1975).
9. M. E. Lesk and B. W. Kernighan, 'Computer Typesetting of Technical Journals on Unix,' Proc. AFIPS NCC **46** (1977), pp. 879-88.
10. *Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer,* Dartmouth College, Hanover, New Hampshire, 1971.

11. R. M. McClure, ‘TMG--A Syntax-Directed Compiler,’ Proc 20th ACM National Conf. (1968), pp. 262-74.

12. S. C. Johnson and B. W. Kernighan, ‘The Programming Language B,’ Comp. Sci. Tech. Rep. #8, Bell Laboratories, Murray Hill NJ (1973).

13. M. Richards, ‘BCPL: A Tool for Compiler Writing and Systems Programming,’ Proc. AFIPS SJCC **34** (1969), pp. 557-66.

14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs NJ, 1978. Second Edition, 1979.

15. D. M. Ritchie, S. C. Johnson, and M. E. Lesk, ‘The C Programming Language,’ Bell Sys. Tech. J. **57** No. 6 (July-August 1978) pp. 1991-2019.