

School of Computer Science and Software Engineering
Monash University

Bachelor of Computer Science Honours (1608), Clayton Campus

Literature Review

Structured process input/output

Reformulating the UNIX command line environment to generate and process XML documents

Cameron McCormack (12793086)

Supervisor: Prof. John Hurst

Table of contents

1. Introduction	3
2. The UNIX command line environment	3
2.1. User programs	4
2.2. Input/output redirection	5
2.3. Command pipelines	6
2.4. The shell	7
3. Conclusion	8
4. Bibliography	8

1. Introduction

The UNIX operating system (Ritchie, Thompson 1974) is an interactive time-sharing system developed in 1969 while its authors were working for Bell Laboratories. It was created with the aim to be an operating system for programmers.

The purpose of an operating system is to manage a computer's resources. Among other tasks, an operating system schedules processes, manages memory and communicates with devices connected to the computer. These services are all provided by the *kernel* of the operating system. What set UNIX apart from other operating systems of the day was the selection of user programs included for manipulating data files. These simple programs were all written to do a specific text processing task well. The ability to compose these programs to create commands that perform more complex tasks is one of the features which has allowed UNIX-like operating systems to remain popular to this day.

UNIX has its roots in the Multiplexed Information and Computing Service (Multics) project (Corbató, Vyssotsky 1965) initiated by MIT, Bell Labs and General Electric. Multics was an ambitious project to develop a large scale, general purpose multi-user operating system. The hope was to implement such features as a virtual memory system (Bensoussan, Clingen, Daley 1972), dynamic linking (Daley, Dennis 1968) and a true hierarchical file system (Daley, Neumann 1965). In 1969, dissatisfied with the progress being made, Bell Labs withdrew from the project. Thompson and his team began to develop a new operating system from scratch. A year of development later, a primitive version of UNIX was up and running.

Even since the early days of UNIX, little research has been done into reworking the command line and even less into the idea that leaving behind the flat text file paradigm would be prudent. This is because keeping data stored in such a simple format has proven simple and powerful. Not all data fits neatly into the record/field organisation of the flat text file, however. It is for this reason that a more structured format for data files and process input and output, such as the Extensible Markup Language (XML) (Bray, et al 2000), needs to be investigated.

In this report the development of important features of the UNIX family of operating systems, in particular its command line environment, are discussed.

2. The UNIX command line environment

The command line environment of UNIX has four main aspects: user programs, input/output redirection, command pipelines and the shell. Together, these provide the user with a powerful means of composing and executing commands to

manipulate data files.

2.1. User programs

The selection of programs available for the user to run is at the heart of the command line environment. From the outset, the focus of user programs was on file manipulation. The first user programs to be written for UNIX were simple file system commands. After having just implemented the file system, Thompson (1979) wrote a few small programs to manage it — programs to let the user create, copy, edit, list and delete files. These commands alone gave the user a simplistic electronic word processor, but not much more. As more user programs were added to the system, three general categories of program emerged: commands which performed a task with no output (such as copying or deleting a file), those which produced output (such as the directory listing program) and those which transformed output (filters, like the program to sort a file's contents).

The user programs that were developed for UNIX were all small programs that performed a task, and only that task, well. Kernighan and Plauger (1976) expounded on that idea and coined the term *software tool*. By writing programs to deal with generalities rather than specifics, focussing them on performing a single task well, the user can build up a toolkit from which they can assemble commands to perform complex tasks.

These programs were all written with a certain data format in mind, namely, flat text. It was held that, to maximise portability and usefulness, data should be stored in plain ASCII (the American Standard Code for Information Interchange — a 7-bit code defining a set of alphabetic, numeric, punctuation and control characters), arranged into fields separated by horizontal whitespace and records separated by newline characters (Gancarz 1995).

While certainly an appropriate design decision for the early 1970s, this file format is not always the best today. ASCII is able to represent only a handful of Western, Latin-based languages. There was not much cause for writing with internationalisation and localisation in mind when the standard UNIX user programs were being created. Today, with global networks such as the Internet, languages other than English have to be considered. A character set such as Unicode (The Unicode Consortium 2000) or the Universal Character Set (UCS) (ISO 10646-1:2000) would certainly solve this problem if they were routinely used and had support from the operating system and its user programs.

The rigid record/field format of flat text files also does not accommodate all types of data. A particular example would be information arranged hierarchically. Naturally such information could be stored in flat text, however an inelegant kludge such as recording a node's depth in the tree would have to be stored in its own field. Far better to keep the relationships between the hierarchical data evident due to the file

format itself. A data file format such as XML could quite easily be used to store such data, being intrinsically hierarchical. XML also has the advantage of using named fields. A user reading an XML document could easily determine the use of certain fields by their names, whereas in a flat text file, such names are absent, and the user would have to rely on some external documentation. To be maximally useful all text generating programs would have to write their output in XML so they could be used in conjunction with XML filtering programs.

2.2. Input/output redirection

One idea which increases the usefulness of the user programs in UNIX is that of input/output (I/O) redirection. General redirection of process input and output was already available in Multics (Corbató, Vyssotsky 1965) and it was from here that the idea was borrowed. One of the tenets of UNIX philosophy that Gancarz (1995) mentions is that all programs should be written as filters. Processes under UNIX have the notion of a *standard input* and *standard output*. If a program has not been given a filename on the command line, the standard input is where a program typically reads its input from. The standard output, analogously, is where a program would write its output to. Any program that reads data from the standard input, transforms it somehow, then writes that data out to the standard output, is considered to be a filter.

Normally, both the standard input and output of a process is the terminal. Thus, program input comes from the keyboard and program output is written to the screen. In UNIX, both the standard input and output of a process can be redirected to some other file or device. With this, a program can be written to handle just a single input source (the standard input) and be able to handle any file implicitly, by having its input redirected before the program is run; similarly for a program's output. This redirection is performed by the *shell*, the command interpreter that elicits commands to run from the user. Programs which did not use the standard input and output, and instead relied on hard coded filenames, were definitely of limited use.

While Multics could also handle such redirection, the method used for enacting the redirection in UNIX was far more elegant. According to Ritchie (1979), Multics required the user to use an explicit command to reassign a program's output. For example, to run the `list` command and redirect its output to a file called `listing`, the following three commands were issued:

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

To accomplish the same task in UNIX, running the `ls` program to generate the listing, just a single command was needed:

```
ls > listing
```

Input redirection could be achieved with a similarly simple command:

```
ed < cmds
```

This command runs the `ed` editor, taking commands from the file `cmds`. The syntax of I/O redirection, using '`<`' and '`>`', has not changed since its original inclusion in UNIX in 1970.

2.3. Command pipelines

The mechanism for composing commands is the *command pipeline*. The concept, which was first seen in the UNIX operating system, is attributed to McIlroy (Ritchie 1979). In 1972 McIlroy proposed the concept of connecting a series of processes together such that the output of one process became the input of the next, the analogy used being that connecting these processes was like screwing together garden hoses. The command pipeline is what enables the reuse of the software tools available to the UNIX user to create complex commands in a single line.

The original syntax used for the pipe concept was similar to the I/O redirection syntax that already existed. A pipeline to sort a file and take the first 10 lines to send to the printer could be written as:

```
sort file >head>lpr
```

Thus, either a file or a command could be placed after a '`>`' character to indicate where the output of the previous command in the pipeline was to go. This syntax, while consistent with the redirection syntax, did allow ambiguous commands to be written. For example, with this syntax there was no way to specify that you wished the output of the `sort` command to be written to a file called `head`, as the shell would determine that `head` was a program and that `sort`'s output should be sent there instead. The syntax was therefore changed to use the vertical bar (or the *pipe symbol* as it has become known). The above command now could be written as:

```
sort file | head | lpr
```

It was at this time, after having implemented the pipeline concept, that the developers of UNIX came to the realisation that this simple extension of I/O redirection was the vehicle for program reuse (Kernighan, Plauger 1976).

When considering the reformulation of the UNIX command line environment to use XML documents, the pipeline concept can remain unchanged. Processes will still write their output to the input of another, only now that output will be in the XML format.

2.4. The shell

The program that provides an interface to the command line environment is the shell. It is this program that repeatedly prompts the user for a command and then executes that command. The shell is perhaps the one aspect of the UNIX command line environment that has undergone a number of changes since its inception.

The original UNIX shell created at the start of 1970 was very simplistic (Ritchie 1979). All the shell did was read a single command from the terminal, load the specified program over the top of itself, and jump to that new program. The operating system relied on the user program to invoke the *exit* system call, which would load a fresh copy of the shell into memory and begin its execution. This was not a true multiprogrammed system, though. Only one process could run at one time.

Subsequent developments to the operating system included the *fork/exec* process control that still exists in today's UNIXes. The Berkeley time-sharing system developed in 1969 already incorporated this method of process control. With this, the UNIX shell could now handle background processes easily. Background processes are those started with a command such as:

```
program &
```

The ampersand indicates that the shell should execute the program in the background and immediately ask the user for another command to run.

From the release of UNIX First Edition in 1971 until the Sixth Edition in 1976, very little changed in the operation of the shell. However, during the development of the Seventh Edition, Bourne (1983) created his own shell, to be known as the Bourne shell. The Bourne shell brought to UNIX shell programming constructs such as iteration and selection that allowed the user to write complex shell scripts. From that point onward, most new shells developed for UNIX, such as the Kornshell (Bolsky, Korn 1995) and the Z shell (Falstad 2001), maintained a syntax compatible with the Bourne shell. One popular shell that was significantly different from the Bourne shell was the C shell, *csh*, written by Bill Joy for inclusion into the Berkeley Systems Distribution of UNIX (McKusick 1999). The C shell, as its name implies, lets the user run commands with a syntax reminiscent of the C programming language (Kernighan, Ritchie 1978).

Attempts to vastly redesign the UNIX shell have been made. One notable design is that of a functional shell, where commands are entered in a form similar to those used in a functional programming language. These shells, such as McDonald's (1987) *fsh* and the Scheme shell (Shivers, et al 2002), *scsh*, allow the user to compose commands as if they were functions.

All of these shells, however, have still been designed with the assumption that the

majority of user programs that are to be composed use flat text for their input and output. The shells can let program output be written verbatim to the terminal, since flat text is eminently human readable. If, however, user programs are to be designed to read and write XML documents, program output is not immediately appropriate for the user to read. XML documents are readable by humans — they do not contain any illegible control codes as some binary formats do — but the information in these documents should be converted into a form that is easy for the user to read. Since the user will be interacting with the shell using a text based terminal, program output destined for the terminal should be transformed into plain text by the shell. Making this final transformation for the user the responsibility of the shell means that user programs do not have to worry about checking whether their output is to be piped into another program, redirected into a file or written to the terminal.

3. Conclusion

None of the UNIX systems developed over the last 30 years have moved away from the flat text file format that the UNIX command line holds to firmly. Gancarz (1995) notes that data files should contain only streams of bytes separated by newline characters. Authors of UNIX systems have thus far adhered to this.

The reason for the success of the flat text file format is its simplicity. It is indeed an easy to use, useful format in which to store data. Thirty years, however, is a very long time in the world of computers, and requirements for data storage have changed. The old adage "if it ain't broke, don't fix it" aside, the requirements for general program input and output format is yet an open area of research.

4. Bibliography

Bensoussan, A., Clingen, C.T. & Daley, R.C. (1972), "The Multics Virtual Memory: Concepts and Design," *Communications of the ACM*, vol. 15, no. 5, pp. 308—318.

Bolsky, M.I. & Korn, D.G. (1995), *The new Kornshell, 2nd ed.*, Prentice Hall, Englewood Cliffs.

Bourne, S.R. (1983), *The UNIX system*, Addison-Wesley, Reading, Mass.

Bray, T., Sperberg-McQueen, C.M., Paoli, J. & Maler, E. (2000, Oct. 6), "Extensible Markup Language (XML) 1.0 (Second Edition)," The World Wide Web Consortium [online], Available: <http://www.w3.org/TR/REC-xml> [Accessed 30 July 2002].

Corbató, F.J. & Vyssotsky, V.A. (1965), "Introduction and Overview of the Multics System," *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 27, no. 1, pp.

185–196.

Daley, R.C. & Dennis, J.B. (1968), "Virtual memory, processes, and sharing in Multics," *Communications of the ACM*, vol. 11, no. 5, pp. 306–312.

Daley, R.C. & Neumann, P.G. (1965), "A General Purpose File System for Secondary Storage," *Proceedings of AFIPS Fall Joint Computer Conference*, vol. 27, no. 1, pp. 213–229.

Falstad, P. (2001, Oct. 24), "The Z Shell Manual," ZSH - THE Z SHELL [online], Available: http://zsh.sunsite.dk/Doc/zsh_a4.ps.gz [Accessed 30 July 2002].

Feiertag, R.J. & Organick, E.I. (1971), "The Multics input-output system," *Proceedings of the Third Symposium on Operating Systems Principles*, pp. 35–41.

Gancarz, M. (1995), *The UNIX Philosophy*, Digital Press, Boston.

International Organization for Standardization (2000), *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*, ISO 10646-1:2000, Geneva.

Kernighan, B.W. & Ritchie, D.M. (1978), *The C Programming Language*, Prentice Hall.

Kernighan, B.W. & Plauger, P.J. (1976), *Software Tools*, Addison-Wesley, Reading, Massachusetts.

McDonald, C.S. (1987), "fsh – A Functional UNIX Command Interpreter," *Software – Practice and Experience*, vol. 17, no. 10, pp. 687–700.

McKusick, K. (1999), "Chapter 2 – Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable," in *Open Sources: Voices from the Open Source Revolution*, DiBona, C., Ockman, S. & Stone, M. (eds.), O'Reilly & Associates, Sebastapol, California, pp. 31–46.

Ritchie, D.M. & Thompson, K. (1974), "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375.

Ritchie, D.M. (1979), "The Evolution of the UNIX Time-sharing System," in *Language Design and Programming Methodology*, Sydney, Australia, September 1979, Springer-Verlag.

Shivers, O., Carlstrom, B.D., Gasbichler, M. & Sperber, M. (2002, May), "Scsh Reference Manual," Scsh - The Scheme Shell [online], Available: <ftp://ftp.scsh.net/pub/scsh/0.6/scsh-manual.ps.gz> [Accessed 30 July 2002].

The Unicode Consortium, (2000), *The Unicode standard, version 3.0*, Addison-Wesley, Reading, Mass.