Unix: Building a Development Environment from Scratch

Warren Toomey

Abstract In April 1969, as part of AT&T's withdrawal from the Multics project, the researchers involved had their "pleasant" development environment taken from them. Bereft of their "toy", the ex-Multics researchers began to cast about for a replacement. Having found nothing suitable, Ken Thompson chose to write one from scratch. By the middle of 1969, he had created a self-hosting operating system on a discarded PDP-7 minicomputer. This was Unix, an operating system whose legacy remains with us today. This paper looks at the creation of Unix after AT&T's departure from the Multics project, the features and innovations in the PDP-7 version of Unix, and the work done in 2016 to restore a working version of PDP-7 Unix from the available source code.

1 The Motivation behind Unix's Development

1969 was not a good year for the Bell Labs researchers at AT&T who were involved in the Multics project. Multics was an attempt to refine and extend many of the contemporary ideas in operating systems (e.g. virtual memory, multitasking) and to build an operating system utility with "a view of continuous operation analogous to that of the electric power and telephone companies" [3].

The Multics designers had set an ambitious list of features for the system, including:

- Virtual, segmented memory, so that a process could access more memory than physically available on the system
- File-mapped memory, for persistent storage of in-memory data structures
- A hierachical filesystem with multiple names for files, symbolic links, quotas on storage and the support for (de)mountable devices

Warren Toomey
The Unix Heritage Society & TAFE Queensland, e-mail: wkt@tuhs.org

- The use of a high-level language to implement the system
- Dynamic linking for executables
- User accounting, administration and security

But Multics was beginning to suffer from Brooks' "second system effect" [2]: it was over-engineered and too ambitious to meet all of these goals on the hardware platform that it was designed for, the General Electric GE-645.

AT&T had joined the Multics project in 1964, along with General Electric and MIT. By April of 1969, with the project behind schedule, AT&T decided to withdraw from Multics. This left the Bell Labs researchers involved in Multics (Ken Thompson, Dennis Ritchie, Doug McIlroy among others) at a loose end. Sandy Fraser describes the situation at Bell Labs in an interview with Mike Mahoney [6]:

> It was quite clear that we were in the course of fairly traumatic change for a lot of people. ... The toy [the GE-645] had gone. The computer room was empty. People were just despondent. Some people were leaving. There was a clear lack of momentum.

Having been burned once with an operating system project, AT&T management were not keen to see further research work done on operating systems. But the researchers themselves had found the Multics system to be a wonderful development environment and they were keen to work on a suitable replacement. Ritchie notes that [4]:

> 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. ... 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.

The Multics researchers at the Labs tried several times to persuade AT&T management to purchase a more modest computing platform on which they would write an operating system, but in retrospect these proposals required AT&T "to spend too much money on too few people with too vague a plan" [12]. Ultimately, all of these plans were rejected.

During this dark period, Thompson was engaged in two activities that would lead to the development of the Unix operating system. The first was the creation of "Space Travel", an accurate simulation of the Sun, planets and moons in the solar system. A player of the simulation could navigate a space ship around the solar system, view the scenery and attempt to land on the planets and moons therein.

"Space Travel" was initially developed on the GE-645 platform, but it was expensive to run ($75 for the CPU time to run one game [12]) and its command-line interface left much to be desired. After the withdrawal from the Multics project, Thompson rewrote "Space Travel" for the PDP-7, a smaller computer at the Labs which had been used for graphical work but which was now outdated and surplus to requirements.

This work introduced Thompson to the PDP-7 platform, one which was severely constrained both in memory (with only 8,192 18-bit words) and disk space (with storage of only 1,024,000 words), and with an obtuse instruction architecture. These restrictions would ultimately influence the design of the Unix system.

2 The Unix Filesystem

The second activity that Thompson was engaged in after the demise of the Multics project was research into filesystem design and structure. Thompson had started this work even before AT&T has formally withdrawn from the Multics project.

> I had built ... a high level simulation of [a] whole file system. It was never down to the point of where you put the addresses, how you expand files and things like that. It was always at some higher level. I think it was just like one or two meetings, Dennis [Ritchie] and [Rudd] Canaday and myself. Was just discussing these ideas of the general nature of keeping the files out of each other's hair and the nitty-gritty of ... where you put block addresses. [5]

From this high-level simulation of a filesystem, Thompson took the PDP-7 and began to implement the filesystem's structure. Initially, this was still a simulation of files and actions on these files, but as time progressed the simulation system was fleshed out into a working operating system.

> To run the file system you had to create files and delete files, re-unite files to see how well it performed. To do that you needed a script of what kind of traffic you wanted on the file system, and the script we had was ... [a] paper tape that said: read a file, read a file, write a file, this kind of stuff. And you'd run the script through the paper tape and it would rattle the disk a little bit. You wouldn't know what happened. You just couldn't look at it, you couldn't see it, you couldn't do anything. [Then] we built a couple of tools on the file system. We used this paper tape to load the file system with these tools, and then we would ... type at the tool that was called a "shell", by the way, to drive the file system into the contortions that we wanted it to measure how it worked and reacted. So [the primitive filesystem] lasted by itself for maybe a day or two before we started developing the [tools] that we needed to load it. [5]

In the 21st century, we see operating systems rightfully as extremely complicated systems, requiring hundreds of developers and thousands of hours to build. In the 20th century, Thompson did not know this. Instead, in the summer of 1969

> My wife went on vacation to my family's place in California to visit my parents. ... She was gone a month to California and I allocated a week each to the operating system [kernel], the shell, the editor, and the assembler, to reproduce itself. During the month she was gone, it was totally rewritten in a form that looked like an operating system, with tools that were sort of known: [an] assembler, an editor and a shell. If not maintaining itself, right on the verge of maintaining itself. [5]

In a month of spare time, Ken Thompson had taken a file system simulator and rewritten it into a self-hosting operating system which we now know as PDP-7 Unix.

2.1 Filesystem Structure

Right from the beginning, the structure of the PDP-7 Unix filesystem had the hallmarks that it would retain for the rest of its life (as shown in Figure 1):

- information nodes (*inodes*) that contain the metadata about each file

- separate directories of filenames, each of which points to an inode, and
- a set of disk blocks that store the contents of each file

The inode could store up to seven disk block numbers, for files up to $7*64 = 448$ words in length. For files bigger than this, the file was marked in the *flags* field as a large file. In this situation, the block numbers pointed to disk blocks that each contained 64 actual disk block numbers, allowing files up to $7*64*64 = 28,672$ words in length.

The *flags* field in the inode also identified the file type as a true file, a directory or a special file, and the permissions on the entry: read and write for the file's owner, read and write for all other users. The owner of a file was identified by the *user-id* field; the concept of user groups would not arrive for a few more years.

One innovation in the PDP-7 Unix filesystem design was the concept of *hard links*. As the name of a file was separated from the metadata in the inode, this allowed two or more filenames to point to the same metadata, as shown in Figure 2.

Each filename linked to the inode was equipotent: no filename was less or more important than any of the other linked filenames. The *numlinks* field in the inode recorded the number of filenames linked to the actual file; only when all filenames were unlinked and the link count dropped to zero was the file deleted from the filesystem.

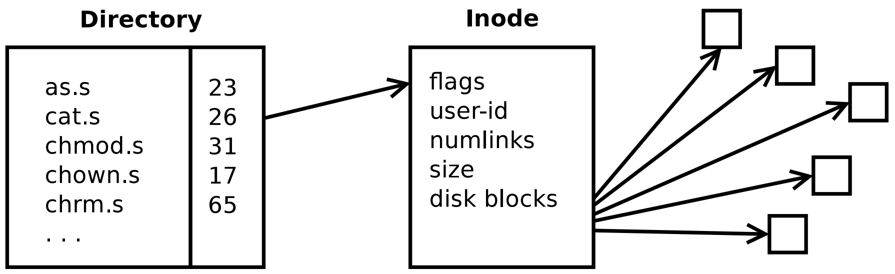

Fig. 1 Inode Structure in PDP-7 Unix

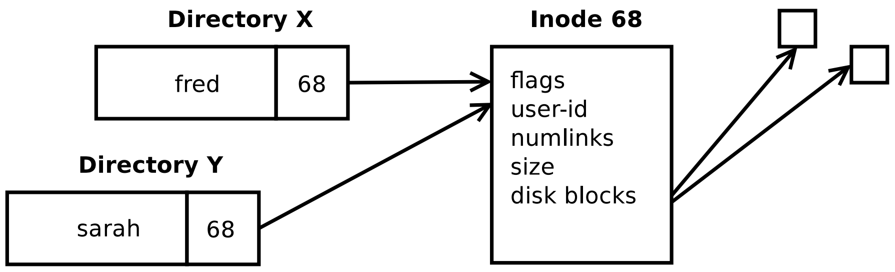

Fig. 2 Hard Links in PDP-7 Unix

2.2 Directories

While the organisation of the files' content and metadata in PDP-7 Unix was fairly well defined from the beginning, the same cannot be said for the organisation of the directories and their relationship to each other.

> The first [filesystem design] was [not a hierarchy of directories but] a directed graph, and we did not restrain it to a tree. We were experimenting with various topologies. [5]

The PDP-7 Unix kernel understood and dealt with the concept of directories (containers for related filenames), and that a directory could contain links to other directories. However, it was agnostic to the arrangement of these directories.

The earliest Unix filesystem, and the one that survives in the restored PDP-7 Unix system, contained two top-level directories, *dd* and *system*, as shown in Figure 3. The *dd* directory contained entries for the system directory and all the user's home directories. The *system* directory contained all the main executable files on the system and the special device files. Today, *dd* would be recognised as the root directory, and *system* as a combination of the */bin* and */dev* (binaries and device files) directories.

To navigate around the filesystem, each directory needed to have an entry back to the *dd* directory. And as the shell was hard-wired to search for executable binaries in the *system* directory, each directory also needed an entry pointing to the *system* directory.

In PDP-7 Unix, the concept of absolute pathnames such as */usr/local/bin/less* did not exist, nor did the concept of relative pathnames (e.g. *../../file* or *subdir1/subdir2/file*). Once a user had moved into a directory, they could only reference files and directories that were visible in that directory.

To obviate this limitation, users could link (add a name) to an existing file so that it was visible in the current directory. Ritchie notes that [12]:

> The *link* operation took the form
>
> ```
> ln 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. [If I was in my home directory *dmr*], to make a link to file *x* in directory *ken*, I might do
>
> | | |
> |---|---|
> | `ln dd ken ken` | Create a link to the *ken* directory in this directory |
> | `ln ken x x` | Create a link to *ken*'s *x* file in this directory |
> | `rm ken` | Remove the now-unneeded link to the *ken* directory |
>
> This scheme rendered subdirectories sufficiently hard to use as to make them unused in practice.

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 [from the *dmr* directory via the *dd* directory] to directory *ken*.

In PDP-11 Unix (written in 1971), the *dd* entry in every directory evolved into the `..` (*dot dot*) entry which points to the directory immediately above it. The extra memory available on the PDP-11 allowed the kernel to support absolute and relative pathnames; once the shell could find executables in the */bin* directory, the *system* entry in each directory was no longer required.

2.3 File Operations and Special Files

PDP-7 Unix implemented a set of file operations that would be immediately recognised by a current Unix or Linux systems programmer:

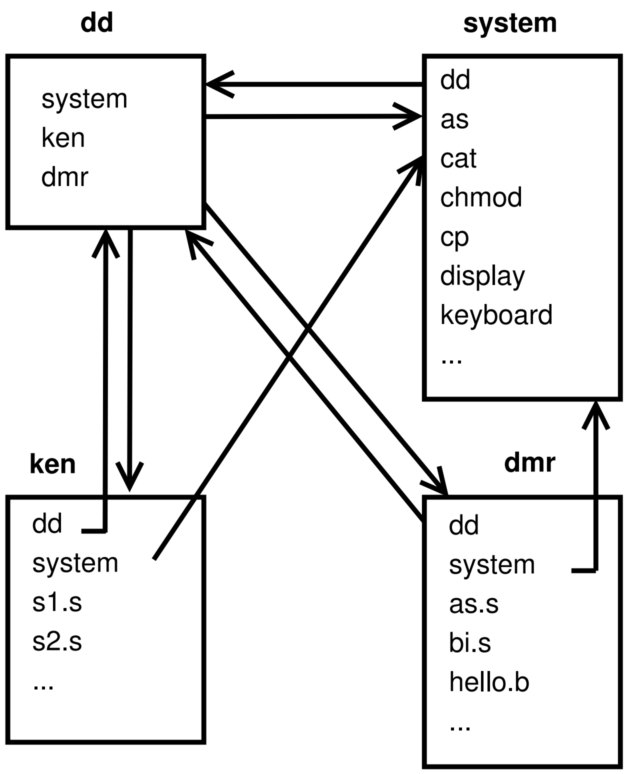

Fig. 3 Directory Structure in PDP-7 Unix

File Operation	**Description**
file descriptor = open(filename, mode)	Open a file read-only or read-write
close(file descriptor)	Close an open file
read(file descriptor, buffer, amount)	Read *amount* words from an open file
write(file descriptor, buffer, amount)	Write *amount* words into an open file
seek(file descriptor, amount, whence)	Move to a specific position in an open file

In this way, the Unix kernel abstracted the storage details of a file and replaced it with a linear array of PDP-7 words (bytes in later Unix versions). PDP-7 Unix also extended this abstraction by introducing the concept of “special files”. Each special file represented the contents of a physical device and could be accessed using the above file operations, regardless of the device’s physical characteristics.

The PDP-7 Unix kernel supported these special device files stored in the *system* directory:

- *ttyin* and *ttyout*, the PDP-7 console device
- *keyboard* and *display*, the Graphics-2 device which was used as a second terminal
- *pptin* and *pptout*, the paper tape device

Thus, on a system with only 8,192 words of memory, PDP-7 Unix was able to provide a multitasking development environment for two users.

3 Processes and Process Control

PDP-7 Unix provided a multitasking environment by dividing the 8K words of memory into two halves. The lower half of memory was reserved for the kernel. The upper half of memory was set aside for the currently running process. PDP-7 Unix swapped processes between memory and disk as part of the context switch between running processes. While this provided process isolation, the PDP-7 hardware did not prevent the running process from accessing the lower 4K words of kernel memory.

What we now know as the canonical Unix set of process control mechanisms (*fork()*, *exec()*, *exit()* and *wait()*) evolved in stages as PDP-7 Unix was developed and then rewritten for the PDP-11 platform.

3.1 Fork and Exec

Before working on Multics and the nascent Unix, Thompson had used the Project Genie research system on the Scientific Data Systems (SDS) 930 computer as a University of Berkeley undergraduate. Project Genie’s main goal was to provide a time-sharing “virtual machine” to a number of users; this included a two-level filesystem where each user had their own directory of files. SDS took the Project Genie system and modified the 930 hardware platform to better support it. The rebadged SDS 940 system became the first commercially successful timesharing system. Later, Xerox Parc would purchase SDS and rebrand it as Xerox Data Systems, and many of the Project Genie researchers would move to Xerox PARC. [17]

Fork() had been developed for Project Genie by Melvin Conway as one of a pair of primitives, *fork* and *join*, to allow processes to be scheduled on a multiprocessor system [7]. The *fork* primitive takes an existing process and creates a second executing process, using the same executable code. The *join* primitive synchronises the two processes that exist after the *fork*, leaving a single process.

Thompson borrowed the *fork()* concept from Project Genie: a PDP-7 Unix process used *fork()* to create an identical copy of itself which could then be scheduled independently. While *fork()* allowed a system to start new processes, they are all identical. Another primitive was needed to allow the system to run programs which are different. In today's Unix, this is the *exec()* mechanism, implemented by the Unix kernel. In PDP-7 Unix, this mechanism was implemented in the shell (as shown in Figure 4).

Once the PDP-7 shell read a command from the user to run a new program, it firstly *fork()*ed a copy of itself; both the original and the new shells now executed concurrently. The original shell waited for the new shell to execute and terminate. The new shell then needed to overwrite itself with the program executable requested by the user.

The execute function in the shell first relocated itself to the top of memory, just under the arguments to the program given on the command line by the user. The relocated function then *open()*ed the executable file and *read()* its contents into the user memory starting at location 4,096. Once the executable had been loaded into memory, the relocated function *close()*ed the file and jumped to location 4,096 to start execution of the new program.

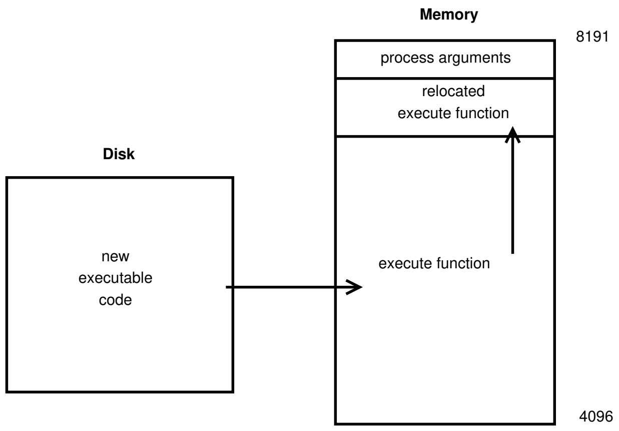

Fig. 4 The shell *exec* mechanism

Ritchie notes that even this simple process control mechanism evolved from a more primitive construct [12]:

> 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 [Project Genie] 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 originally 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 I/O 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* mechanism would have been considerably more complicated, if only because *exec* as such did not exist; its function was already performed, using explicit I/O, by the shell.

3.2 Exit and Wait

In today's Unix systems, a process which has *fork()*ed a new process can *wait()* for that process to terminate. The new process can *exit()* to terminate its execution, and this returns an exit status value back to the original process which is *wait()*ing.

As with *fork()* and *exec()*, *wait()* and *exit()* evolved from other mechanisms. In this case, the previous mechanisms were more, not less, sophisticated.

> The primitives that became *exit* and *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 [*exit* and] *wait* calls of the present Unix 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 [12].

4 Utilities in Unix

Not only was PDP-7 Unix a versatile operating system whose kernel was squeezed into 4K words of memory, its developers also created a number of important utilities whose descendants are still in use today.

In Ritchie's "Draft of the Unix Timesharing System" [11], written in 1971 when both PDP-7 and PDP-11 Unix systems were in existence, he documented the commands and utilities listed below. Only a few of these would be unfamiliar to a current Unix user.

Command	**Description**
as	The assembler
b	The B compiler
cat	Concatenate files
chdir	Change the current directory
chmod	Change a file's permissions
cp	Copy a file
db	The debugger
ed	The text editor
ln	Link a new filename to a file
mkdir	Make a directory
mv	Rename (move) a file
nm	List the symbols in an executable
pr	Print a file
rm	Remove a file or directory
roff	The text processor
sh	The shell
tm	Print time-related kernel information
un	List unidentified symbols in an executable

Several of the utilities in PDP-7 Unix deserve a fuller coverage.

4.1 The Shell

PDP-7 Unix was the first operating system to provide a command-line user interface which was replaceable by the user. Earlier systems such as CTSS and Multics had command-line interpreters, but these were part of the "system" and could not be changed or modified.

PDP-7 Unix was also the first system that provided "standard input" and "standard output" abstractions to its utilities. In essence, every process started with one already open file for reading input, and another open file for sending output. By default, these open files were connected to the user's terminal.

The PDP-7 Unix shell provided mechanisms for the user to redirect these open files to existing (or new) files, and also to special files. Ritchie notes that [12]:

> the very convenient notation for I/O 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 I/O redirection mechanism [*author's note, see [8]*] embodying named I/O 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, 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 [the file] *xx*, on Multics the notation was [12]
>
> ```
> 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 I/O 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 [8] lists *iocall* as an 'author-maintained,' that is non-standard, command.) Because both the Unix I/O 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.

Pipes, another important Unix concept, would not arrive in the Unix system until the 3rd Edition of PDP-11 Unix in 1973.

4.2 Text Processing with roff

One of the earliest uses of the Unix system, apart from research into operating systems, was as a document processing system. The evolution of the document processing tool, *roff*, is interesting. It began life as J. Saltzer's *runoff* tool, written in assembly language for the CTSS operating system [15]. This had been rewritten by Doug McIlroy in BCPL for the Multics operating system. In turn, this had been rewritten as *roff* (with reduced functionality) in PDP-7 assembly for PDP-7 Unix. Then, to justify the purchase of a PDP-11 minicomputer, the Unix researchers rewrote *roff* into PDP-11 assembly language to support the document processing needs of AT&T's Patent department. Current Linux users might regard "open source" as a relatively new concept, but the sharing and development of source code has been an important mechanism from the beginning of computing.

4.3 The B Compiler

As noted earlier, Thompson had written an assembler for the PDP-7 minicomputer to make the Unix system self-hosting. The kernel and all the original utilities in PDP-7 Unix were written in PDP-7 assembly. But the Unix developers saw the need for higher level languages. Thompson notes that [12]:

> [We wanted to write the system in a high-level language] right from the start... That was a Multics influence. And just the complexity of maintaining the thing, we just knew that, you can't maintain something [in assembly, let alone] write it and get it going, [as] it will evolve. ... PL/I [the high-level language used on Multics] was too high for us, or even the simpler version of PL/I in Multics (a thing called EPL). After [PDP-7 Unix] was up, or, simultaneous with Unix coming out, BCPL was just emerging and that was a clear winner with both of us [i.e. Thompson and Ritchie]. Both of us were really taken by the language and did a lot of work with it.

BCPL was a word-oriented systems language developed by Martin Richards at Cambridge University [10], which was designed to be somewhat portable to other machines and systems. As BCPL was still too "big" a language for the PDP-7, Thompson took the essential structures from BCPL and wrote a bytecode interpreter on the PDP-7 for an intermediate language which implemented these structures. He then wrote a compiler that targeted this intermediate language, and created the B language.

> It was the same language as BCPL, but it looked completely different. Syntactically it was, you know, a redo, [but the] semantics were exactly the same as BCPL. And in fact the syntax of it was, if you didn't look too close, you would say it was C. Because in fact it was C, without types [12].

Very little of the use of the B language in Unix survives: just a few simple PDP-7 programs written in B. The B language, and its compiler, would evolve through a number of stages into the C language [13]. Ritchie and Thompson would rewrite the

Unix kernel in the high-level C language in 1974 [14], thus achieving their goal to write the system in a high-level language.

5 Restoring the PDP-7 Unix System

For many years, the PDP-7 Unix system was essentially a mythical beast. Although its existence was documented [11, 16], none of the source code had survived except the *dsw* command, posted by Dennis Ritchie to the *net.unix-wizards* Usenet newsgroup in 1984.

As the founder of the Unix Heritage Society,[1] I had spent a lot of time recovering old Unix systems and restoring them to working order. Notable successes were the first edition of PDP-11 Unix and the first C compiler [19]. But the PDP-7 Unix system remained elusive.

In 2016, a long-time member of the Unix Heritage Society revealed that he possessed a printout of some of the source code to PDP-7 Unix, which he had copied in the 1980s when he worked in the Research labs at AT&T. He was strongly encouraged to scan this in and make it available to the Society.

The source code to any computing system is, by itself, useless unless there is an environment to convert the system into machine executables, and then an environment to run these executables. In 2016, the PDP-7 was a distant memory and, even if one was available, there was no assembler to convert the source code into executable format.

Fortunately, Bob Supnik and a cohort of excellent developers had built SimH [18], a simulator of many early computer systems including the PDP-7 and its peripherals. Once the PDP-7 Unix source code became available, a team of three (Phil Budne, myself and Robert Swierczek) began the task of converting the source code into a working system.

The first task was to write a new PDP-7 assembler, and this was started by myself and completed by Phil Budne. We did have the source code for the PDP-7 Unix assembler, but this was a “chicken and egg” situation where the assembler source could not yet assemble itself.

With our assembler we could assemble source code into PDP-7 machine code but not yet execute the machine instructions. We could not use SimH, as this simulates a whole system; to execute any Unix code, we needed a full system: machine, kernel, working and populated filesystem, and utilities. A bug in any of these would prevent the entire system from working correctly.

Instead, we chose to implement a “user mode” simulator: one which implements most of the PDP-7 machine instructions, and converts PDP-7 Unix system calls into system calls to the underlying host OS. An example of a similar “user mode” simulator is Wine [1]. Using this simulator, we could test our assembler and the

[1] http://www.tuhs.org

source code to the original PDP-7 Unix utilities without worrying about the Unix kernel or filesystem.

With confidence in the assembler and the utilities, we turned our attention to the PDP-7 Unix kernel and the construction of a file system, i.e. the layout of inodes, directories and file blocks. Phil Budne took on the task of getting the kernel to work, and I wrote a tool to build a suitable filesystem. Both tasks were arduous. The Unix developers had left very few comments in their assembly code, and PDP-7 instructions were foreign to all members of our team. We spent a lot of time deducing the purpose of sections of assembly code, and adding comments and annotations to it.

The original tool to construct a PDP-7 Unix filesystem had been lost, and a replacement tool needed to be written. For myself, the complete lack of documentation on the layout of the filesystem made the construction of a filesystem with directories and our compiled binaries difficult. Using what information we had at hand (the kernel source code, documentation for the PDP-11 Unix system and the ability to single-step instructions in SimH), I was eventually able to write a Perl script that builds a working PDP-7 Unix filesystem. The script takes as input a text file that describes the files and directories to be placed on the filesystem, the ownership and permissions on the files and directories, and the links between the various directories. Using this information, the script generates a filesystem image that the PDP-7 Unix kernel can understand.

Phil Budne had similar difficulties with the PDP-7 Unix kernel. There was no documentation on the Graphics-2 device, which had been built in-house at AT&T. Not only did Phil have to deduce its modes of operation, he also had to add code to SimH to simulate this device.

Ken Thompson had taken a month in 1969 to write the kernel, the assembler, the shell and the debugger. Of these four crucial tools, the source code to the shell had gone missing. Using the information provided by Ritchie in "The Evolution of the Unix Timesharing System" [14], along with snippets of extant PDP-7 source code, Phil Budne was able to heroically reconstruct a working shell for the PDP-7 Unix system. Of the 525 lines of code in the reconstructed shell, only 14 lines of code were borrowed from the original PDP-7 Unix *init* source code.

As the work on all of these threads finally came together, the team was able to announce that the PDP-7 Unix timesharing system had been returned to full working order; the original source code and the tools we used in the reconstruction are available for download on Github[2]. This includes several utilities for which the original source code has gone missing: *cp*, *ln*, *ls* and *mv*.

In the background, Robert Swierczek had been working on the reconstruction of the B compiler. Its source code also had been lost, but the bytecode interpreter for the intermediate language had survived. Robert used the source code to the earliest C compiler, removed the code dealing with types, and retargetted it to produce code for the bytecode interpreter. In a masterpiece of reverse "bootstrapping", Robert wrote the B compiler so that it was at the same time valid B and C code, and so that it could recompile itself. To build the compiler, first it is compiled using a modern

[2] https://github.com/DoctorWkt/pdp7-unix

C compiler[3]. This not-yet-PDP-7 compiler is then used to compile the B compiler source code (again), producing the intermediate language version which can be run using the B interpreter on PDP-7 Unix.

PDP-7 Unix Element	**Original LOC**	**Newly-written LOC**
Kernel	2,812	0
Editor	1,290	0
Assembler	1,026	576
Shell	unknown	525
Text Processor	unknown	780
Utilities	5,930	1,340
Filesystem creator	unknown	339
B Compiler	unknown	825

6 Conclusion

The development of Unix was, in some ways, a reaction against the "bigger is better" mentality of Multics. Thompson felt that [5]:

> It was a good idea that we were getting out of Multics. That it was too big, too expensive, too over-designed. It was just clear it was an exercise in building monstrosities.

But Unix was not solely a rejection of the Multics concept: indeed, many of the ideas from Multics were simplified and added to Unix. Ritchie notes that [4]:

> We were a bit oppressed by the big system mentality. Ken wanted to do something simple. So Unix wasn't quite a reaction against Multics, it was more a combination of these things. Multics wasn't there for us any more, but we liked the feel of interactive computing that it offered. Ken had some ideas about how to do a system that he had to work out; and the hardware available as well as our inclinations tended to trying to build neat small things, instead of grandiose ones.

In time, the Unix operating system would espouse a design philosophy which has been summarised by Doug McIlroy [9]:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

With the PDP-7 version of Unix, Thompson, Ritchie and others were still experimenting with the concepts and structures that would ultimately develop and crystallise into this philosophy. Here, they are still grappling with the filesystem structure, the essential Unix process mechanisms, the features of the shell and the ability to abstract device operations into generic file operations. The final two elements that would provide Unix with its "toolbox" philosophy, pipes and a portable kernel, would arrive in the following four years.

[3] Many warnings are issued at this stage.

Using an outdated hardware platform, the PDP-7, and with only 8,192 words of memory, Thompson and Ritchie were able to build a multitasking, multiuser operating system with a multi-directory filesystem. PDP-7 Unix would not only distill and simplify many of the concepts from other systems (Multics, Project Genie), but it would also introduce innovations such as hard links, devices represented as special files and generalised I/O redirection. Many of the features, utilities and system calls introduced in PDP-7 Unix are still in use today on its BSD descendants as well as clean-room rewrites such as Linux. The legacy of this tiny operating system lives on, nearly 50 years after its birth.

References

1. Amstadt, B., Johnson, M.K.: Wine. Linux Journal **1994**(4es), 3 (1994)
2. Brooks, F.P.: The mythical man-month : essays on software engineering. Reading, Mass. Addison-Wesley Pub. Co. (1975)
3. Corbató, F.J., Saltzer, J.H., Clingen, C.T.: Multics: the first seven years. In: American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972, pp. 571–583 (1972). DOI 10.1145/1478873.1478950
4. Mahoney, M.: Interview with Dennis Ritchie (1989). URL `http://www.tuhs.org/Archive/Documentation/OralHistory/transcripts/ritchie.htm`
5. Mahoney, M.: Interview with Ken Thompson (1989). URL `http://www.tuhs.org/Archive/Documentation/OralHistory/transcripts/thompson.htm`
6. Mahoney, M.: Interview with Sandy Fraser (1989). URL `http://www.tuhs.org/Archive/Documentation/OralHistory/transcripts/fraser.htm`
7. Nyman, L., Laakso, M.: Notes on the history of fork and join. IEEE Annals of the History of Computing **38**(3), 84–87 (2016). DOI doi.ieeecomputersociety.org/10.1109/MAHC.2016.34
8. Project MAC: The Multiplexed Information and Computing Service: Programmers' Manual. Mass. Inst. of Technology, Cambridge MA (1969)
9. Raymond, E.S.: The art of Unix programming. Addison-Wesley Professional (2003)
10. Richards, M.: BCPL: A Tool for Compiler Writing and System Programming. In: Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, AFIPS '69 (Spring), pp. 557–566. ACM, New York, NY, USA (1969). DOI 10.1145/1476793.1476880
11. Ritchie, D.M.: Draft: The UNIX Time-sharing System (1971). URL `http://www.tuhs.org/Archive/PDP-11/Distributions/research/McIlroy_v0/UnixEditionZero-Threshold_OCR.pdf`
12. Ritchie, D.M.: The Evolution of the Unix Time-Sharing System. In: Proceedings of a Symposium on Language Design and Programming Methodology, pp. 25–36. Springer-Verlag, London, UK (1980)
13. Ritchie, D.M.: The Development of the C Language. In: The Second ACM SIGPLAN Conference on History of Programming Languages, HOPL-II, pp. 201–208. ACM, New York, NY, USA (1993). DOI 10.1145/154766.155580
14. Ritchie, D.M., Thompson, K.: The UNIX Time-sharing System. Commun. ACM **17**(7), 365–375 (1974). DOI 10.1145/361011.361061
15. Saltzer, J.H.: Manuscript Typing and Editing, 2nd edn., p. AH.9.01. MIT Press (1965)
16. Salus, P.H.: A Quarter Century of UNIX. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1994)
17. Spinrad, P., Meagher, P.: Project Genie: Berkeleys piece of the computer revolution (2009). URL `http://www.coe.berkeley.edu:80/news-center/publications/forefront/archive/forefront-fall-2007/features/berkeley2019s-piece-of-the-computer-revolution`

18. Supnik, B., Walden, D.: The Story of SimH. IEEE Annals of the History of Computing **37**(3), 78–80 (2015)
19. Toomey, W.: The Restoration of Early Unix Artifacts. In: 2009 USENIX Annual Technical Conference (USENIX ATC 09). USENIX Association, San Diego, CA (2009)