An Overview of Unix

Jehan-François Pâris

Computer Science Department
University of Houston
Houston, TX 77204-3475

1. CHRONOLOGY

1969 First version of Unix written by K. Thompson for a PDP-7 and later rewritten by K. Thompson and D. Ritchie in C.

1973 Kernighan and Ritchie present Unix at first SOSP

1976 Version 6, running on a PDP-11; is the first version to be widely available outside Bell Labs.

1977 First Berkeley Software Distribution (for the PDP-11).

1978 Version 7, the base version of all future AT&T releases.

1979 Virtual memory implementation done at U. C. Berkeley for the VAX 11/780.

1980-... Various releases of Berkeley Unix: 4.2 BSD was the first to support TCP/IP.

1982 System III, the first commercial distribution of Unix by AT&T's Unix Support Group, which became later Unix System Laboratories (USL).

1983 Eighth edition by Bell Labs with stream I/O.

1983 Various releases of System V marketed by AT&T's Unix Support Group.

mid 80's A. Tanenbaum starts working on MINIX, a didactic version of Unix running on a 8086.

1985 MACH, a "new kernel for Unix development" from CMU, later to be used as a base for NEXTSTEP and OSF/1.

1987 Ninth edition by Bell Labs.

1989 Plan 9 from Bell Labs (not Unix-compatible).

1991 Linus Torvalds releases the first usable version of Linux on October 5 (0.01)

1992 AT&T sells USL to Novell

1993 4.4 BSD, the last release of Unix from U. C. Berkeley.

1994 Linus Torvalds releases Linux 1.0

1994 4.4 BSD-Lite a "freely redistributable" Unix that is free of any AT&T code.

1996 Linus Torvalds releases Linux 2.0

2001 Apple releases Unix-based MAC OS X

2007 Google unveils Android.

2011 The first chrome laptop arrives.

Four major Unix traditions have co-existed:

1. Bell Labs (purely research),
2. AT&T (System V and derivatives),
3. Berkeley (includes ULTRIX) and OSF/1, and
4. Linux.

BSD survives in two free incarnations, FreeBSD and BSDLite.

2. KEY FEATURES

Unix started as an attempt to scale down Multics, the giant OS project of the late sixties that involved at one time the MIT, General Electric and AT&T. It was designed by programmers for programmers and used simple solutions whenever feasible. It had no virtual memory. Process sizes in the earlier versions were limited to 64K, the size of the PDP-11 address space. As a result, Unix has evolved along a toolbox approach: large tasks are normally accomplished by combining existing small programs rather than developing complex subsystems.

Almost from the beginning, Unix was written in a high-level language and its source code made available to licensees, which explains the numerous ports and variants. The fact that the command interpreter—or shell—is a separate program has allowed it to be easily modified or replaced.

The Unix file system is one of Unix most durable legacies. While its file naming mechanism based on a hierarchy of directories and sub-directories was borrowed from MULTICS, the inclusion of devices in this hierarchy was an innovation.

3. THE FILE SYSTEM

Unix files can be *ordinary files*, *directories* or *special files*. Ordinary files are string of uninterpreted ASCII or non-ASCII characters (bytes). Physical record boundaries are invisible to the users. Five basic file operations are implemented:

`open()` returns a file pointer,

`read()` reads a given number of bytes,

`write()` writes a given number of bytes,

`lseek()` moves the byte pointer to a new byte offset,

`close()` closes a file.

All reading and writing are sequential. The effect of direct access is achieved by manipulating the offset through **`lseek()`**.

Unix maintains a file cache in main memory and buffers all its file I/O: This delayed write policy increases the I/O throughput but will result in lost writes whenever a process terminates in an abnormal fashion or the system crashes. Terminal I/Os are also buffered one line at a time, which makes it difficult to read anything that is not terminated by a return.

Directories are tables that map directory entries with physical file addresses. These addresses are local to the current disk partition and represent the index number (*i-number*) of the file in that disk partition table. Hence directory subtrees cannot cross disk partition boundaries unless a new *file system* is mounted somewhere in the tree. The whole directory structure is constrained to have the form of a rooted subtree although non-directory files may appear in several directories (within the same file system). Berkeley Unix has *symbolic links* that can bypass these restrictions. They are also called soft links because they are nothing but aliases for the "real" path name.

Special files are directory entries associated with I/O devices. Since device names have the same syntax as regular file names, any program expecting a file name as parameter can be passed a device name instead.

3.1. Protection

Unix specifies three classes of users (owner, all members of one of the groups specified in **`/etc/groups`**, all users) and three access rights (read, write and execute).

To allow users to execute programs accessing data whose access needs to be controlled, Unix has a *set user ID* mechanism: any file that has its set user ID flag set will execute as if it was executed by its owner. It is the responsibility of the owner of the file to ensure that any other user cannot modify the file (with all the possible consequences).

Earlier versions of Unix had no provision for locking files. Process that needed to synchronize the updates of a file needed to use a *lock file*. (A process attempting to update the file was trying to create an empty read-only file with a predefined name. The lock was released by removing that file.)

System V allows file and record locking at a *byte-level* granularity through **`fcntl()`**. Berkeley Unix has an *advisory* file lock mechanism: locks are only enforced for processes that request them.

3.2. Version 7 Implementation

Each disk partition contains a *superblock* containing the parameters of the file system stored in the disk partition, an *i-list* that has one *i-node* for each file or directory in the file system and a free list. Pointers to i-nodes are called *i-numbers*.

Each *i-node* contains:

1. the user-ID and the group-ID of the owner of the file,
2. the file protection bits (including the set user-ID bit),
3. the physical disk addresses of the file contents,
4. the file size,
5. times of creation, last usage and last modification,
6. the number of directory entries pointing to the file, and
7. a flag indicating if the file is a directory, an ordinary file, or a special file.

The remainder of the disk is divided into fixed size blocks or pages. The original block size *b* was 512 bytes but larger block sizes are more prevalent now.

To facilitate the management of the i-table, all i-nodes have a fixed-format with enough space for only 13 block addresses. For non-special files, the 10 first addresses point at the 10 first blocks of the file. For

larger files, the 11th address points to an *indirect block* containing the addresses of *b*/4 additional blocks and the 12th address to a *double indirect block* containing the addresses of *b*/4 indirect blocks, each containing the addresses of *b*/4 data blocks. Hence a total of $10+b/4+b^2/16$ blocks can be addressed. If required, the 13th address can point to a *triple-indirect* block.

Since i-nodes are loaded into memory at file open time, the first ten blocks of a file can always be accessed directly. The cost of accessing the other blocks of a file is greatly reduced by the system block buffering mechanism, which will tend to keep in main memory the most recently used indirect blocks. Another important effect of this block buffering mechanism is the fact that the return from a write call often takes place before the data have been actually transferred from the buffer to the disk.

3.3. Berkeley Improvements

Since there was one single free list per file system, consecutive blocks of a file tended to be allocated to random cylinders. Selecting the proper block size was also a problem because small block sizes lead to inefficient I/O while large block sizes cause too much internal fragmentation (45.6% wasted space for 4 Kbytes blocks vs. 6.9% for 512 byte blocks).

The Berkeley *Fast File System* (FFS) groups the cylinders of each disk partition into groups of consecutive cylinders. Each *cylinder group* contains (a) a redundant copy of the superblock, (b) space for i-nodes and (c) a bit map of all available blocks in the cylinder group. This information is placed at a different offset on each cylinder group to protect against failures of individual platters. FFS attempts to keep consecutive blocks of the same file in the same cylinder group as the file i-node.

FFS eliminates the third level of indirection by increasing the minimum block size from 512 bytes to 4 kilobytes. To fight internal fragmentation, file blocks can be partitioned into 2, 4, or 8 fragments that can be used to store small files or the tails of larger files. Note that these two last optimizations require that the file system rarely become 100% full.

Other changes include (a) a device-specific method for computing the "rotationally optimal" position for the next block of a file, (b) a provision for longer file names and (c) the addition of two extra block addresses in the i-node, making it possible to address directly the first 12 blocks of a file.

4. PROCESS CREATION

Unix processes can share their program text segments but have their own address spaces. Processes are created through the **`fork()`** system call.

```
int pid
 ...
 if ((pid = fork()) == 0) {
     /* child process */
     ...
     exec(...);
     _exit(1) /* error */;
}
while (pid != wait(0));
/*parent waits*/
```

When a **`fork()`** is executed, the process executing the **`fork()`** splits itself into two independently executing processes having independent copies of the original address spaces and sharing all opened file descriptors. In one of the processes, called the *child*, the value returned by **`fork()`** is zero while the value returned to the other process—the *parent*—is the process-id of the child process. Copying virtual address spaces being a costly operation, Berkeley Unix has a **`vfork()`** primitive that shares the same address space between parent and child until the child process does an **`exec()`**. A *copy-on-write* mechanism would have been a better solution.

Note that the Unix **`wait()`** primitive waits for the completion of any child of the calling process but returns the process-id of the terminating child process.

5. INTERPROCESS COMMUNICATION

The earlier versions of Unix had only limited interprocess communication mechanisms, namely *pipes* and *signals*. IPC between unrelated processes was mostly done through the use of temporary files.

Pipes allow related processes to communicate between themselves using the same **`read()`** and **`write()`** primitives that are used for I/O. The call **`pipe(fildes)`** returns two file descriptors **`fildes[0]`** and **`fildes[1]`** such that any bytes written on **`fildes[1]`** can be read on **`fildes[0]`**. The mechanism is heavily used by Unix to build applications from *filters*.

A filter is a program that normally reads its input from its standard input (**`stdin`**) and writes its output on its standard output (**`stdout`**).

The shell utilizes the **|** symbol to specify that the standard output of a program is to become the standard input of the next one, as in

```
tbl textfile | eqn | troff
```

The major limitation of the mechanism lies in the fact that pipes are only passed from parents to children.

Signals are another mechanism to communicate between processes; receiving processes may specify what action to take upon the receipt of a signal using signal(). If no action is specified, the process will terminate. The signal SIGKILL cannot be caught or ignored.

5.1. System V IPC

The Unix System V IPC package provides

1. message queues through **msgget()**, **msgsnd()**, **msgrcv()** and **msgctl()**;
2. shared memory through **shmget()**, **shmat()** and **shmdt()**;
3. FIFOs, which are named pipes and are created through **mknod()**; and
4. semaphores through **semget()**, **semop()** and **semctl()**.

Refer to the System V Programmer's Reference Manual for more details. (The **semop()** call interface is a great example of a bad user interface.)

5.2. Berkeley IPC

Berkeley systems provide a single mechanism for local IPC and for network communication using sockets. Sockets are end points of communication. Sockets have normally an address attached to them and the nature of this address depends on the communication domain of the socket. Two domains are currently implemented under Berkeley Unix, namely the Unix domain (**AF_Unix**) in which addresses share the format of ordinary file system pathnames (as in **/usr/paris/example**) and the Internet domain (**AF_INET**), which obeys the DARPA Internet conventions. Unix domain sockets are currently restricted to IPC between processes on the same machine.

Sockets can be of various types, which correspond to different classes of services. The three most important classes of services currently implemented are:

1. **SOCK_STREAM**, which provides reliable duplex sequenced byte streams and guarantees that no data will be lost or duplicated; message boundaries are not preserved;
2. **SOCK_DGRAM**, which transfers in either directions messages of variable sizes and does not guarantee that messages will arrive in sequence or that they will not be lost or duplicated;
3. **SOCK_RAW**, which allows direct access to the underlying protocols.

Sockets are created using the **socket()** system call, which returns a socket descriptor. Sockets that need to be addressed by another socket must have a name bound to it using the **bind()** system call. Other processes can then attempt to initiate a connection with it using the **connect()** system call to connect their local socket with the remote socket.

The Berkeley IPC was specifically designed to implement as efficiently as possible the client/server model of interprocess communication. Server processes use the **listen()** system call to notify the kernel that they are ready to accept **connect()** calls from clients and use the **accept()** call to accept a given connection request from a client To allow client requests to overlap, **accept()** returns a new socket descriptor that defines the server's end of the client/server connection. This socket descriptor can be passed to a new child of the server process, which remains ready to wait for new connection requests. Ordinary **read()** and **write()** system calls can be used to communicate between the client and the server's child servicing it. The connection can be closed using the **close()** call; **shutdown()** instructs the system about the proper disposal of pending messages at closing time.

SOCK_DGRAM sockets also allow individually addressed messages using the **sendto()** and **recvfrom()** system calls.

6. PROCESS SCHEDULING

The Unix schedulers belong to the general class of multilevel feedback queues. Ready processes are assigned a scheduling priority that determines in which run queue they are placed. To select which process to run, the scheduler scans the run queues from highest to lowest priority and chooses the first process on the first non-empty queue. If several processes reside in the same queue, the system runs them *round robin*, i. e. giving to each of them a quantum of the CPU before putting them back to the end of the queue. 4.3 BSD used a time quantum of 100 ms; this figure being an empirical compromise between the two objectives of maximizing throughput while keeping an acceptable response time for interactive jobs such as editors.

The scheduling priority of a process is a function of a base priority (**PUSER**), its current CPU usage (**p_cpu**)and a user-settable weighting factor (**p_nice**: see **nice(1)**).

In System V,

p_usrpri = PUSER+p_cpu/2+p_nice

and **p_cpu** is updated every second according to a decay function,

decay(p_cpu) = p_cpu/2.

In Berkeley Unix,

p_usrpri=PUSER+p_cpu/4+2×p_nice

and **p_cpu** is updated every second according to

p_cpu=(2×ld×p_cpu)/(2×ld+1)+p_nice

where **ld** is a sampled average of the length of the run queue over the last minute.

System V Release 3 assigns fixed priorities to real-time processes and variable priorities to time sharing processes have variable priorities managed by multi-level feedback queues. It also lets the system administrator can modify all the parameters of these queues. While the solution is outdated, it provides an excellent example of how multilevel feedback queues work.

Linux 2.4 used a fair scheduler that partitions CPU time into *epoch*s. At the beginning of each epoch, each process was assigned a *time quantum* that specified the maximum CPU time the process can have during that epoch.

Processes that exhausted their time quantum were not allowed to get additional CPU time until the next epoch started. Processes that released the CPU before their time quantum was exhausted could not get more CPU time during the same epoch. Each epoch ended when all ready processes had exhausted their time quanta.

Process priorities were the sum of their base priority plus the amount of CPU time left to the process before its quantum expired.

While simple and elegant, this policy lacked scalability, and could not handle well real-time systems. In addition, it was not designed for multi-core processors.

Linux 2.6 uses O(1) schedulers whose speed does not depend on the number of active processes.

7. MEMORY MANAGEMENT

Memory management was somewhat neglected in the earlier versions of Unix because the system was then running on machines with very small address spaces and no memory mapping facilities. O. Babaoglu and W. Joy did the first virtual memory implementation of Unix on a VAX 11/780 in 1979. It has been adopted by most modern implementations of Unix.

One of the major problems that had to be faced was the lack of *page referenced bits*—or use bits—in the VAX architecture. This limitation precluded the usage of many interesting page replacement policies such as sampled working set, page fault frequency or Multics' Clock-1 second chance policy. VMS had avoided the problem by using a two-level hybrid page replacement policy combining some aspects of FIFO and least recently used. VMS assigns to each process a fixed size partition called the process' *working set* and uses a FIFO replacement policy to manage it. Pages that are expelled from a working set are put at the end of a large global queue of pages waiting to be expelled from main memory. These pages are also marked invalid in the page tables of their respective processes. Every time a page fault occurs, VMS scans first this global queue before attempting to bring the missing page from secondary storage. The mechanism approximates LRU quite well provided that (a) enough pages are allocated to the process working sets to keep the number of false page faults under control, and (b) the global queue is large enough to allow the rescue of pages that had been erroneously expelled from the working set of their process.

These two conditions were found to be especially difficult to achieve in the Unix systems that were typical of the late seventies as Unix creates a non-trivial amount of short-lived processes whose memory requirements are hard to predict. Babaoglu and Joy decided instead to simulate the page referenced bit by software: each page has a *software valid bit* and a *software page referenced bit* in addition to its *hardware valid bit*. Whenever the referenced bit needs to be reset, the software resets the hardware valid bit at the same time. The next reference to the page will result in a page fault and the page fault handler will use this opportunity to set the hardware valid bit and the software page referenced bit. The major limitation of the mechanism was the cost of the interrupts required to set the hardware valid bit and the *software page referenced bit* (250 μs on a VAX 11/780). Babaoglu and Joy selected therefore a page replacement policy minimizing the number of page referenced bit sets and resets. They implemented a

version of the MULTICS Clock-1 page replacement policy with the following modifications:

1. the *speed* at which the hand of the clock sweeps the circular list of active pages was limited to 300 pages/sec. (which guarantees that the software simulation of page referenced bits will never take more than 10% of the total CPU time), and
2. the policy tries to maintain a *pool of free pages* instead of expelling pages on demand.

Babaoglu and Joy quickly found that the 512-byte page size of the VAX was too small. As a result, they decided to use *clusters* of physically adjacent hardware pages to simulate a larger page size. A limited amount of *prepaging* was added to the fetch policy, mostly as a result of the decision to manage clusters of pages. A mechanism allowing to turn off the page referenced bit simulation when executing processes exhibiting anomalous non-local behaviors was also quickly added.

This implementation was more recently updated to take into consideration the larger memory sizes of more recent Berkeley Unix installations. A second hand, following the original clock hand at a fixed angle, was added to reclaim faster pages that have been marked not referenced.

8. THE LEGACY OF UNIX

The major contribution of Unix to the field of operating systems and the one that will have the lasting impact in the future is the demonstration that operating systems did not need to be architecture specific. Previous operating systems were normally written in assembly language and were always designed for a given target architecture. Unix is almost entirely written in C and most of the code makes no assumptions about the underlying hardware. As a result, it has been successfully ported to a wide range of architectures going from home computers to mainframes and Cray supercomputers. MacOS X is based on a Mach/BSD kernel. Both Android and Chrome run on the top of customized versions. While Microsoft had access to Unix even before they developed MS-DOS, Windows NT and its successors were more influenced by VMS than Unix.

It is difficult to overestimate the impact that Unix portability has had and continue to have. First, Unix users discovered opportunity to switch manufacturers without having to retool their whole operation to a new OS. The existence of a portable OS also had a dramatic impact on the hardware industry. Established manufacturers tried first to ignore Unix as they were deeply committed to proprietary operating systems and were used to rely on these to generate customer loyalty. Upstart companies and microchip makers saw the things differently: Unix made it possible to experiment with instruction set design and to move from architectures privileging hand written assembly code to architectures tailored to execute as fast as possible C code. The move to Reduced Instruction Set Computers would have been much slower without Unix. The overall outcome has been a much more competitive marketplace and a much faster pace in hardware innovations.

Unix success has also led to an industry-wide effort to establish standards for user and system interfaces so that softwares could be more easily ported from an OS architecture to another. These standard interfaces tend to follow the lines of the corresponding Unix interfaces and might end being the most lasting legacy of Unix.

There are many other features of Unix, which made a lasting impact on other operating systems. Unix popularized the concept of a hierarchical file system that is largely device transparent. Unix also legitimized the use of the command language as a bona fide programming language by introducing the proper control structures, as well as output redirection and pipes.

Like all large-scale human endeavors, Unix has also betrayed some of its original tenets. While the early versions of Unix were good illustrations of the "small is beautiful" philosophy, this paradigm has been progressively put aside as successive versions of Unix became more powerful and more complex.

References

1. M. J. Bach, *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs (1986)
2. O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits," *Proc. 8th ACM Symposium on Operating Systems Principles*, (1981), pp. 78-86.
3. IBM DeveloperWorks, *Inside the Linux 2.6 Completely Fair Scheduler: Providing fair access to CPUs since 2.6.23*, http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/index.html (retrieved August 23, 2016)
4. S. Bourne, "Unix Time-Sharing System: The Unix Shell," *The Bell System Technical Journal*, 57, 6 (1978), pp. 1971-1990.
5. S. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the*

4.3 BSD Unix Operating System. Addison-Wesley, Reading (1989).

6. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for Unix," *ACM Transactions Computer Systems*, 2, 3 (1984), pp. 181-197.
7. M. K. McKusick, G V. Neville Neil, *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley (2004)
8. E. Nemeth, G. Snyder, S. Seebass, Trent R. Hein, Unix *System Administration Handbook*. 2nd Edition, Prentice Hall (1995).
9. J. S. Quarterman, A. Silberschatz, and J. L. Peterson, "4.2 BSD and 4.3 BSD as Examples of the Unix System," *ACM Computing Surveys*, 17, 4 (1985), pp. 379-418.
10. D. Ritchie and K. Thompson, "The Unix Time-Sharing System," *The Bell System Technical Journal*, 57, 6 (1978), pp. 1905-1929.
11. D. Ritchie, "Unix Time-Sharing System: A Retrospective," *The Bell System Technical Journal*, 57, 6 (1978), pp. 1947-1969.
12. D. Ritchie "The Evolution of the Unix Time-Sharing System," *The Bell System Technical Journal*, 63, 8 Part 2 (1984), pp. 1577-1593.
13. R. W. Stevens, Unix *Network Programming*, Prentice Hall Software Series (1990).
14. K. Thompson, "Unix Time-Sharing System: Unix Implementation," *The Bell System Technical Journal*, 57, 6 (1978), pp. 1931-1946.
15. M. Rochkind, *Advanced Unix Programming*, 2nd edition, Prentice-Hall, Englewood Cliffs (2004).
16. U. Vahalla, Unix *Internals: The New Frontiers*, Prentice-Hall (1996)