SIGINFO
Everything Is a Punch Card

SIMSON L. GARFINKEL

Simson L. Garfinkel is a senior computer scientist at the US Census Bureau and a researcher in digital forensics and usability. He recently published *The Computer Book* (Sterling, 2019), a coffee table book about the history of computing.
sigmail@simson.net

It's commonly said that in UNIX, "everything is a file."

The meaning of this catchy aphorism is that most UNIX resources can be accessed using names in the file system with a small, consistent set of function calls. So not only can we `open()`, `read()`, and eventually `close()` regular files like `/etc/motd` and `/bin/ls`, we can read the contents of the hard drive (if you have suitable permissions) by opening `/dev/disk0`, the first physical disk on a Macintosh, and even `/dev/mem`, the Linux "device" that lets user processes read system memory.

In this column I'll look at the origins of files and file systems, and contrast the UNIX approach with a subtly different approach that was developed for the Multics operating system, in which files are actually named segments in a two-dimensional memory address space. On Multics, saving a "file" was really creating a named memory segment and then persisting it to long-term storage. Finally, I'll look at how the idea of named persistent memory segments backed by non-volatile memory is making a comeback and will likely be an important part of the storage stack in the near future.

The Historical File

Back in the 1500s a *fyle* was a string or wire used to bind together paper documents, or so reports the *Oxford English Dictionary:* "Thapothecaries shall kepe the billis that they serue, vpon a fyle" (1525). Also spelled *file*, by 1600 the word was used variously to denote the documents in a legal proceeding; a catalog, list, or roll; or even the figurative thread of a person's life.

Put simply, English has had a difficult relationship with the word "file" since the beginning. Sometimes the word refers to the case or container for organizing physical embodiments of information, sometimes it refers to the objects put into that container, and sometimes it refers to the information itself.

Although these days most information that's stored in files is video, when I think of a "file" on my computer, I typically think of a text file. That is, I think of a collection of lines, each somewhere between 1 and 80 characters long, separated by some kind of "end-of-line" character. And for this I have to thank Herman Hollerith and the company he created, The Tabulating Machine Company.

Hollerith graduated from Columbia University in 1879 and took a job working for one of his professors, William P. Trowbridge, who had just taken a temporary assignment working on the 1880 Census in Washington, DC, where he was compiling statistics on power and machinery used in manufacturing. Looking back, this wasn't very surprising: by all accounts Hollerith was a hard-working, brilliant, and ambitious fellow who frequently attracted the mentorship of his older colleagues.

Once in Washington, DC, Hollerith met another future mentor, John Shaw Billings, a surgeon who had become the director of the Army Surgeon General's library after serving in the Civil War. Billings was also working on the Census, where he was in charge of tabulating vital statistics.

The 1880 Census, also known as the 10th Census, was a massive information operation. Census employees collected data from all over the country and brought it to Washington, DC, where it was manually processed according to many different criteria—a process called tabulating—and eventually published. You can think of this processing as a series of SQL SELECT statements with suitable GROUP BY and WHERE clauses. The 10th Census used computers as well, but they were all the human kind [1].

The Card File

Billings suggested that Hollerith create some kind of machine to mechanize the laborious tabulation process. Perhaps Hollerith could build a machine that counted notches on cards of paper, Billings suggested, with each card representing a single person's demographic characteristics, like their age or sex? Hollerith found this idea fascinating and eventually transferred to work under Billings in the vital statistics division just to spend a few months learning the job. When the work on the 10th Census started winding down in 1881, Hollerith moved to Boston, where he had been offered a teaching position at the Massachusetts Institute of Technology.

When he wasn't teaching, Hollerith experimented with ideas for the census machine. Inventing was far more interesting to Hollerith than teaching—he couldn't stand the thought of teaching the same course a second time—so he quit the Institute and took a job back in Washington, DC as a patent examiner. But once he learned the ins-and-outs of the US patent system, he quit *that* job and became a full-time inventor, supporting himself by doing patent work for others.

Hollerith's first census machine patent application described a machine with a long tape of paper and rows of holes representing each person, but Hollerith eventually returned to an idea suggested by Billings. He built prototype machines and, with Billings' help, used them in vital statistics projects in Baltimore and New York City.

The 11th Census had a competition for a machine to assist in the tabulations: Hollerith's machine was one of three tested. Hollerith won the contract, supplied the tabulating equipment for the 11th Census, and eventually incorporated The Tabulating Machine Company in 1896. The company merged with its competitors in 1911 to form the Computing-Tabulating-Recording Company, which was renamed International Business Machines in 1924.

The Hollerith cards used in the 1890 Census had 12 rows of 24 columns and were sized $6\frac{5}{8}$″ by $3\frac{3}{4}$″ so that they fit perfectly inside boxes used to store paper money. When users needed more storage per card, the space between the rows was reduced, allowing the card to hold 45 columns. This still wasn't enough storage, so in 1928 IBM standardized on a card of $7\frac{3}{8}$″ by $3\frac{1}{4}$″ with rectangular holes punched in 80 columns of 12 rows. That was the final standard, and it had lasting influence. The IBM 3270 display terminal introduced in 1971 had an 80-character wide screen, as did the IBM PC introduced in 1981. Indeed, *PEP 8—Style Guide for Python Code*, last revised in August 2013, recommends that source code not exceed 79 characters because some editors wrap when the user tries to edit the last character on an 80-character line.

Older readers may recall receiving punch card checks and utility bills imprinted with the words "do not fold, spindle or mutilate." A *spindle* is a nasty spike pointed straight up and mounted on a weight for holding papers. That is, a spindle is a fyle, and you should avoid using a spindle to file your card file, because the extra hole will be read as an error.

Larger punch cards were used for voter ballots in various parts of the United States until the election of November 2000, after which they were largely replaced due to concerns over their usability and accuracy.

The Circular File

Punch cards were all the rage in information processing for more than a half century. The US Social Security Administration had a master card file sorted by each person's nine-digit social security number. It had another set of punch cards sorted according to the phonetic code of each person's surname. Chrysler had punch cards for its inventory control system. Grades from standardized exams were punched onto cards [2], making it easier for researchers to compute statistics. Really, almost every bit of information that was needed for later processing was stored on punch cards. Even though early computers had magnetic tape, data on tapes was frequently loaded using high-speed punch card readers, and put back onto cards for long-term storage after processing.

In 1956, IBM announced the IBM 305 RAMAC, the Random Access Memory Accounting System. The system's breakthrough technology was the IBM 350, the world's first commercial hard drive. There were 50 metal disks, each with 100 concentric tracks, and a moving read/write-head assembly. The whole thing could store five million 6-bit characters, or 3.75MB. The base system rented for $3,200/month, of which $650 was for the disk storage unit. IBM sold more than a thousand of these vacuum tube-based computers until 1961, when the line was discontinued.

Programming the 305 was complicated: not only was there nothing resembling a modern file system, the program itself had to include pauses to allow the RAMAC's disks to rotate into the appropriate position and for the head to complete any required seek operations. When I downloaded and read the 1957 manual [3], I was most surprised by the matter-of-fact way that IBM described the 305.

It's not a general-purpose computer that has a first-in-the-world megabyte-sized random access memory: it's a system designed for the specific task of helping companies automate inventory, billing, and accounts receivable. That is, it's an electronic punch card file! The big paradigm shift that the manual tries to convey to the reader is that idea that "files are located *in the machine,*" —emphasis in the original—rather than in some external box.

Old paradigms die hard.

On UNIX, Everything Is a File

Modern UNIX and Linux owes much of its flexibility to the way that the operating system handles files and file systems. While other operating systems maintain a different namespace for every physical device, UNIX puts everything into a single hierarchy, a single unified naming system for all files currently accessible.

The second advantage of the "everything is a file" approach manifests when programs running on UNIX get a "file" to open and, lo, it's actually the name of a device. Most UNIX programs will still work, provided that the calling process has the correct authorization to open the file.

This ability to treat devices as files extends to pseudo-devices like `/dev/stdin`, `/dev/stdout`, and `/dev/tty`, which map to `stdin`, `stdout`, and the controlling terminal of the current process. For example, while some programs like `wc` will take their input from `stdin` if no input file is provided, other programs will only take their input from "files." You can give these programs `/dev/stdin` as their input file and then put them into a shell pipeline, like a properly written UNIX program.

Recently, I had a program that decided what file type it was reading by looking at the file's extension. I wanted the program to read its input from a pipe. My solution was to create symbolic in `/tmp` with the appropriate extension, point the link at `/dev/stdin`, and give the program the link for its input. Convoluted, perhaps, but the hack worked the first time.

Another thing that is obviously not a file is memory. Yes, Linux systems have devices like `/dev/mem` and `/dev/kmem` that let programs access memory through the file system, but memory is not file. And although UNIX and Linux have the `mmap()` family of system calls to map files into memory or write blocks of memory out to disk, use of these calls is quite limited. That's an unfortunate result of the UNIX "flat" memory model, in which the program's code, data stack, and any "extra" information can all be accessed using the same pointers.

Because UNIX processes only have access to that single flat address space, files mapped into memory might be mapped into a different location each time a program runs—and it certainly is mapped into different locations when run in different programs. This isn't a problem when code is mapped into memory, as is the case with shared libraries, because most shared code is compiled as position independent code (PIC).

Loading nondeterministically into different regions of memory is a big problem when loading data, however. After all, the whole reason to map a disk file into memory is speed. But if the program can't guarantee where the file is going to land, then the program will need to resort to using indirect memory accesses and various kinds of pointer arithmetic to find every data object. Such approaches are now so well-established that we accept them without much thought, but having to mediate practically every memory reference with pointer arithmetic can have a significant performance impact.

The flat memory space of modern operating systems also has security and reliability implications: many security problems of the last three decades ultimately result from the fact that a `(char *)` pointer in the C programming language can effectively reference any part of the executing program's data, stack, or code.

Multics Files Are Segments

Many of the ideas that make UNIX and Linux great were developed for Multics, the project started in 1965 by MIT's Project MAC, Bell Telephone Laboratories, and General Electric Company's Large Computer Products Division [4]. For example, the very idea of a single tree-structured, hierarchical file system holding all of the system's programs and user files was invented for Multics. Also invented for Multics is the idea that the command processor—the Multics creators called it a shell—would be a normal user program, and that commands would be implemented as programs sitting in the file system rather than making commands a privileged part of the operating system.

Files certainly existed at the time that the Multics project started, as did virtual memory, which was invented in 1962 for the Atlas computer at the University of Manchester. But Multics unified files and memory in a way that was not widely adopted.

On Multics, files are simply pieces of memory that are given names in the hierarchical file system. Multics uses the word *segments* to describe these pieces of memory.

A Multics process might have hundreds of segments mapped into memory at any given time. When segments are mapped into a process context—called loading—the segment's symbolic file system name is mapped to a segment number. Pointers are confined within a segment. Corbató and Vyssotsky's paper from the 1965 Fall Joint Computer Conference [5] describes this as "two-dimensional" addressing. Segments make it easy to provide for the secure sharing of code and data between users, because a single segment can be accessed concurrently by any number of processes, while the underlying hardware controls whether an individual process can read or write to each specific segment.

Multics ran on the GE-645, a computer created for the purpose of running Multics. The actual hardware is somewhat odd by current standards. The GE-645 had a 15-bit segment number and an 18-bit offset to a word within the segment; the underlying system used 36-bit words, divided into four 9-bit "bytes." This machine still runs today, albeit in emulation. You can log into a community Multics system and try it out at https://www.ban.ai/multics/.

Segments neatly circumvent the problems of shared, persistent memory: with two-dimensional pointers (segment and offset), the offset of an individual datum doesn't change when it is mapped out and mapped back in. This means that Multics didn't need to use position independent code, didn't need to relocate code when it was loaded into memory, and allowed code to be shared between executing programs, which meant that only a single copy of each library needed to be loaded into memory—something that wasn't widely available in the UNIX world until the 1990s.

Intel tried to implement segments on the iAPX 432 in 1975, but the project was overly ambitious and ran late. So instead, the company focused on the 8086, a 16-bit version of its successful 8080 microprocessor. Launched in 1978, the 8086 has just four "segment" pointers—the code segment, the data segment, the stack segment, and the "extra" segment—and a 20-bit address is computed by taking a 16-bit segment number, shifting it to the left 4 bits, and adding the offset. That is, segments were a tool for extending memory from 64 KiB to 256 KiB, but not for managing data, shared libraries, or implementing memory protection.

The modern x64 architecture still has these CS, DS, ES, and SS pointers, but they are all set to 0 (zero) to create a flat 64-bit memory space. Now 64 bits is a lot of addressable memory, and we could use some of them for some kind of virtual segment number, but on today's hardware only 48 bits of the address pointers are used: take away 16 bits for a segment number, and that leaves only 32 bits for an offset within a segment. So it might be possible to implement something like Multics segments on modern hardware, but it ultimately won't deliver the same security properties that Multics did because Multics segment/offset pointers simply could not overflow into the next segment. Still, a segmented memory model might be an improvement over what we have today—provided that the segments were large enough.

The Next File

The idea of saving memory in named segments may be coming back into vogue with the advent of so-called storage-class memory (SCM). This memory is a lot like the magnetic core memory of the 1950s and '60s in that it is directly addressable from the CPU and doesn't forget its contents when it is turned off. It's faster than disk and more expensive per byte than disk or flash, but slower and less expensive per byte than DRAM.

One such memory system currently on the market is Optane, manufactured by Micron for Intel. You can buy Optane packaged on a DIMM module or as a PCIe card that looks like a SSD. Plug it into a DIMM slot, and Optane looks like slow memory that doesn't get reset after restart—but be careful, because your system's power-on self-test (POST) might wipe it unless the POST is programmed not to do so. Plug Optane into a PCIe slot, and it looks like an incredibly fast, but small, SSD.

SCM memory is here today, and it might open up a lot of possibilities if people would simply use it. For example, you can buy *today* a server with 24 DIMM slots and give it 12 2-TiB Optane modules, for 24 TiB of non-volatile memory, and 12 128-GiB DDR4 modules, for 1.5 TiB of main memory. You could use such a system to build a massive database server: keep the index and transaction log in the 24 TiB Optane storage, and you won't need to flush the index to disk when the server shuts down and read it into memory when the server starts up. Bailey, Ceze, Gribble, and Levy explored other ideas for using SCM in their 2011 HotOS XIII paper, "Operating System Implications of Fast, Cheap, Non-Volatile Memory"[6]. Meanwhile, Yang, Kim, Hoseinzadeh, Izraelevitz, and Swanson explore the performance of Optane in their FAST '20 paper, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory" [7].

References

[1] For a brief description of the use of human computers in the 10th Census, see https://bit.ly/slg-100. For a more detailed study, see David Alan Grier's excellent book on the subject, *When Computers Were Human* (Princeton University Press, 2005).

[2] To learn more about the use of punch cards in the testing of Indian children, see K. E. Anderson, E. G. Collister, and C. E. Ladd, *The Educational Achievement of Indian Children: A Re-Examination of the Question: How Well Are Indian Children Educated?* (Bureau of Indian Affairs, 1953). https://bit.ly/slg-101.

[3] It's not hard to find IBM RAMAC 305 manuals online. My favorite is http://ed-thelen.org/comp-hist/22-6264-1-IBM-305-RAMAC-ManualOfOperation.pdf.

[4] For more information about Multics, see https://multicians.org/history.html.

[5] F. J. Corbató, and V. A. Vyssotsky, "Introduction and Overview of the Multics System," in *Proceedings of the November 30—December 1, 1965, Fall Joint Computer Conference, Part I*, pp. 185-196: https://www.multicians.org/fjcc1.html.

[6] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating System Implications of Fast, Cheap, Non-Volatile Memory," in *Proceedings of 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011: https://www.usenix.org/legacy/events/hotos11/tech/final_files/Bailey.pdf.

[7] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST '20)*: https://www.usenix.org/conference/fast20/presentation/yang.