Virtual Filesystems: what they are and why we need them

Introduction

Linux early contributor and author Robert Love authoritatively states, “A *filesystem* is a hierarchical storage of data adhering to a specific structure.” This description applies equally well to VFAT, git and the NoSQL database Cassandra. What then is it that distinguishes a filesystem?

Filesystem basics

The Linux kernel's requirement is that in order to be a filesystem, an entity must in addition implement the open(), close(), read() and write() methods on persistent objects that have names associated with them. From the point of view of object-oriented programming, the kernel treats the generic filesystem as an abstract interface, and these big-four functions are "virtual", with no default definition. Accordingly the kernel's default filesystem implementation is called a "virtual filesystem" (VFS).

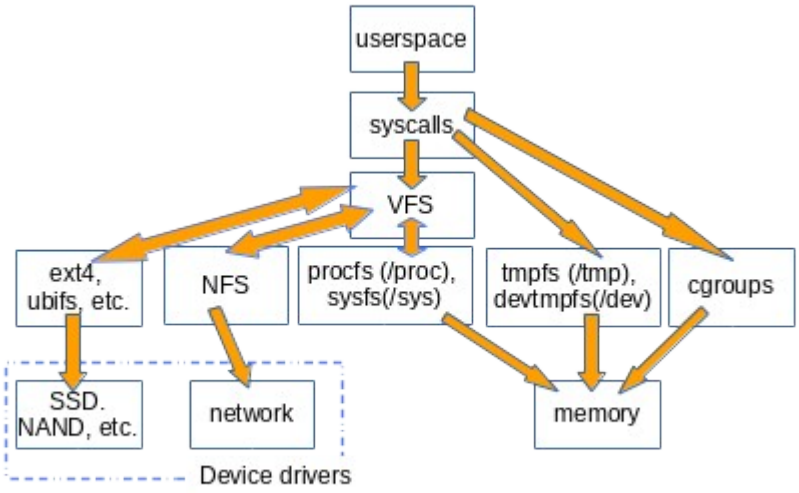
```
$ file /dev/console
/dev/console: character special (5/1)
$
$ ls -l /dev/console
crw------- 1 root root 5, 1 Jan 18 11:34 /dev/console
$
$ echo "hello world" > /dev/console
hello world
$
```

If we can open(), close(), read() and write(), it is a file as this console session shows.

VFS underly the famous observation that, in Unix-like systems, "everything is a file." Consider how weird it is that the tiny demo above featuring the character device /dev/console actually works. The image shows an interactive bash session on a virtual teletype (what "tty" abbreviates). Sending a string into the virtual console device makes it appear on the virtual screen. There are other even odder properties of VFS. For example, it's possible to seek in them.

The familar filesystems like ext4, NFS and /proc all provide definitions of the big-four functions in a C-language data structure called "file_operations." In addition, particular filesystems extend and override the VFS functions in the familiar object-oriented way. As Robert Love points out, the abstraction of VFS enables Linux users to blithely copy files to and from foreign operating systems or abstract entities like pipes without worrying about their internal data format. On behalf of userspace, via a system call, a process can copy from a file into the kernel's data structures with the read() method of one filesystem, then use the write() method of another kind of filesystem to output the data.

The function definitions that belong to the VFS base type itself are found in the fs/*.c files in kernel source, while the subdirectories of fs/ contain the specific filesystems. The kernel also contains filesystem-like entities such as cgroups, /dev and tmpfs, which are needed early in the boot process and are therefore defined in the kernel's init/ subdirectory. cgroups, /dev and tmpfs do not call the file_operations big-four functions, but rather directly read from and write to memory. The diagram below roughly illustrates how userspace accesses various types of filesystems commonly mounted on Linux systems. Not shown are constructs like pipes, dmesg and POSIX clocks that also implement struct file_operations and whose accesses therefore pass through the VFS layer.

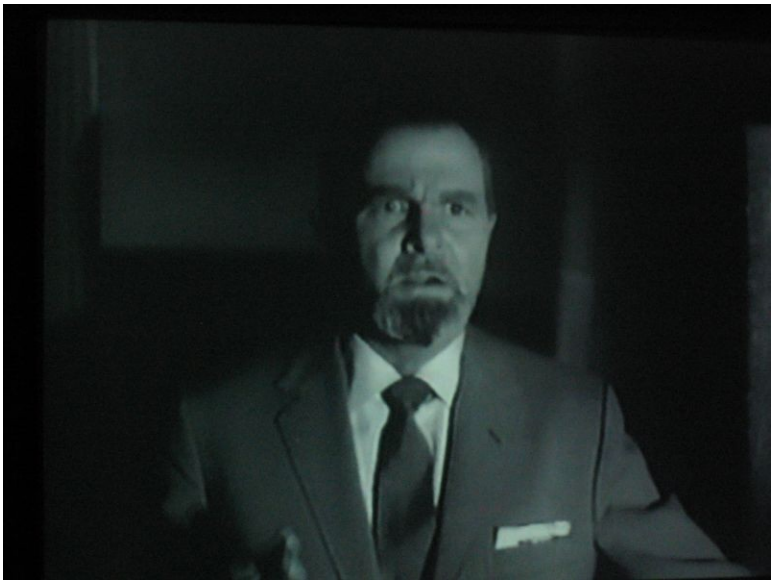


VFS are a "shim layer" between system calls and implementors of specific file_operations like ext4 and procfs. The file_operations functions can then communicate either with device-specific drivers or with memory accessors. tmpfs, devtmpfs and cgroups don't make use of file_operations but access memory directly.

The existence of VFS promotes code reuse, as the basic methods associated with filesystems need not be reimplemented by every filesystem type. Code reuse is a widely accepted software engineering best practice! Alas, if the reused code introduces serious bugs then all the implementations that inherit the common methods suffer from them.

/tmp: A simple tip

An easy way to find out what VFS are present is a system is to type "mount | grep -v sd | grep -v :/", which will list all mounted filesystems that are not resident on a disk and not NFS on most computers. One of the listed VFS mounts will assuredly be /tmp, right?



Everyone knows that keeping /tmp on a physical storage device is crazy! Credit: https://tinyurl.com/ybomxyfo

Why is keeping /tmp on storage inadvisable? Because the files in /tmp are temporary (!), and storage devices are slower than memory, where tmpfs are created. Further, physical devices are more subject to wear from frequent writing than memory is. Lastly files in /tmp may contain sensitive information, so having them disappear at every reboot is a feature.

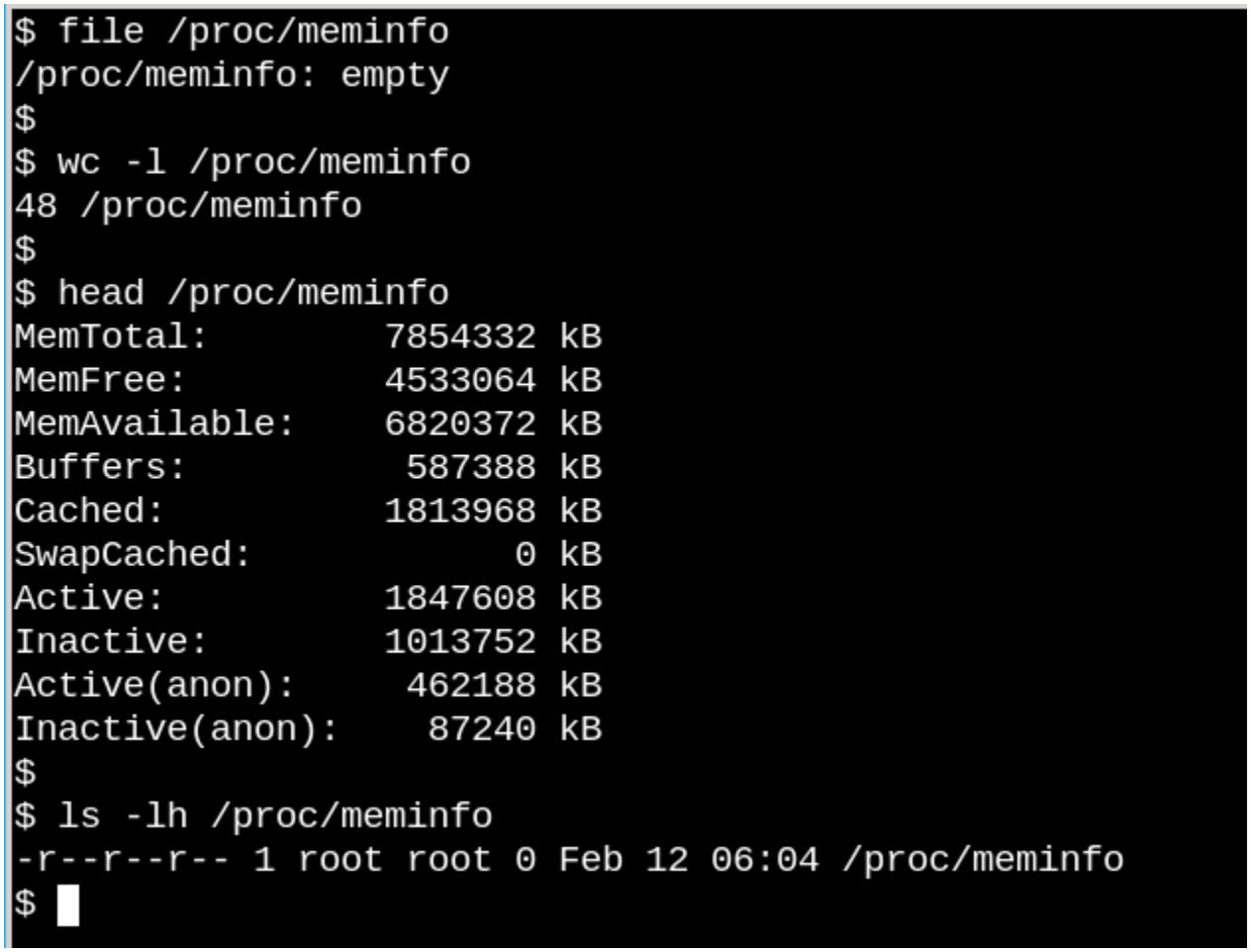
Unfortunately, installation scripts for some Linux distros still create /tmp on a storage device by default. Do not despair should this be the case with your system. Follow simple instructions on the always excellent Arch Wiki to fix the problem, keeping in mind that memory allocated to tmpfs is not available for other purposes. In other words, a system with a gigantic tmpfs with large files in it can run out of memory and crash. Another tip: when editing the /etc/fstab file, be sure to end it with a newline, as otherwise your system will not boot. Guess how I know.

/proc and /sys

Besides /tmp, the VFS with which most Linux users are most familiar will be /proc and /sys. (/dev relies on shared memory and has no file_operations). Why two flavors? Let's have a look in more detail.

The procfs offers a snapshot into the instantaneous state of the kernel and the processes that it controls for userspace. In /proc the kernel publishes information about the facilities it provides like interrupts, virtual memory and the scheduler. In addition, /proc/sys is where the settings that are configurable via

the sysctl command are accessible to userspace. /proc/<PID> directories are where status and statistics on individual processes are reported.



/proc/meminfo is an empty file that nonetheless contains valuable information.

The behavior of /proc files illustrates how unlike on-disk filesystems VFS can be. On the one hand, /proc/meminfo contains the information presented by the command 'free'. On the other hand, it's also empty! How can this be? The situation reminds me of a famous article written by Cornell University physicist N. David Mermin in 1985 called "I*s the moon there when nobody looks? Reality and the quantum theory*." The truth is that the kernel gathers statistics about memory when a process requests them from /proc, and that there actually *is* nothing in the files in /proc when no one is looking. As Mermin said, "It is a fundamental quantum doctrine that a measurement does not, in general, reveal a pre-existing value of the measured property." The answer to the question about the moon is left as an exercise.

The files in /proc are empty when no process accesses them.

The apparent emptiness of procfs makes sense, as the information that is available there is dynamic. The situation with sysfs is different. Let's compare how many files of at least one byte in size there are in /proc versus /sys:

```
$ find /proc -type f -size +1c 2>/dev/null
/proc/config.gz
$
$ find /sys -type f -size +1c 2>/dev/null | wc -l
12736
```

procfs has precisely one, namely the exported kernel configuration, which is an exception since it only needs to be generated once per boot. /sys, on the other hand, has lots of larger files, most of which comprise one page of memory. sysfs files typically contain exactly one number or string in contrast to the tables of information produced by reading files like /proc/meminfo. The purpose of sysfs is to expose the readable and writable properties of what the kernel calls "kobjects" to userspace. kobjects' only purpose is reference-counting: when the last reference to a kobject is deleted, the system will reclaim the resources associated with it. Yet /sys constitutes most of the kernel's famous "stable ABI to userspace" which no one may ever, under any circumstances, "break." That doesn't mean that files in sysfs are static, which would be contrary to reference-counting of volatile objects. The kernel's stable ABI instead constrains what *can* appear in /sys, not what is actually present at any given instant. Listing the permissions on files in sysfs gives an idea of how the configurable, tunable parameters of devices, modules, filesystems, etc. can be set or read. Logic compels the conclusion that procfs is also part of the kernel's stable ABI, although the kernel's Documentation doesn't state so explicitly.

```
# ls /sys/block/sda/
alignment_offset   events            inflight    removable  sda7/       uevent
bdi@               events_async      integrity/  ro         size
capability         events_poll_msecs mq/         sda1/      slaves/
dev                ext_range         power/      sda2/      stat
device@            hidden            queue/      sda5/      subsystem@
discard_alignment  holders/          range       sda6/      trace/
#
# ls -l /sys/block/sda/removable
-r--r--r-- 1 root root 4096 Feb 11 17:37 /sys/block/sda/removable
#
# cat /sys/block/sda/removable
0
```

Files in sysfs describe exactly one property each for an entity and may be readable, writable or both. The "0" in the file reveals that the SSD is not removable.

Snooping on VFS with eBPF and bcc tools

The easiest way to learn how the kernel manages sysfs files it to watch it in action, and the simplest way to watch on ARM64 or x86_64 is to use eBPF. eBPF or "extended Berkeley Packet Filter" consists of a virtual machine running inside the kernel that privileged users can query from the command line. Kernel source tells the reader what the kernel *can* do; running eBPF tools on a booted system shows instead what the kernel actually *does*. Happily, getting started with eBPF is pretty easy via the bcc tools, which are available as packages from major Linux distros and have been amply documented by Brendan Gregg. The bcc tools are Python scripts with small embedded snippets of C, meaning that anyone who is comfortable with either language can readily modify them. At this count there are 80 Python scripts in bcc/tools, making it highly unlikely that a system administrator or developer cannot find an existing one relevant to her/his needs.

To get a very crude idea about what work VFS are performing on a running system, try the simple vfscount or vfsstat, which show that dozens of calls to vfs_open() and its friends occur every second.

```
# sudo ./vfsstat.py
TIME         READ/s  WRITE/s CREATE/s   OPEN/s  FSYNC/s
19:45:49:        57       14         0       16         0
19:45:50:        14       15         0        1         0
19:45:51:        14        6         0        3         0
19:45:52:       146       48         0        4         0
19:45:53:       159       29         0        3         0
19:45:54:       156       28         0        0         0
^C#
```

vfsstat.py is a Python script with an embedded C snippet that simply counts VFS function calls.

For a less trivial example, let's watch what happens in sysfs when a USB stick is inserted on a running system.

```
$ ### 0. Simple example: type the following command, then insert a USB stick:
$ sudo ./trace.py 'p::sysfs_create_files'
PID     TID     COMM            FUNC
7       7       kworker/u16:0   sysfs_create_files
^C
$
$ ### 1. Complex example: repeat USB insertion but with a more complex command:
$ sudo ./trace.py -K -I /usr/src/linux-source-4.19/include/linux/sysfs.h 'p::sys
fs_create_files(struct kobject *kobj, const struct attribute **ptr) "Created fil
ename is %s", (*ptr)->name'
PID     TID     COMM            FUNC             -
7711    7711    kworker/u16:3   sysfs_create_files Created filename is events
        sysfs_create_files+0x1 [kernel]
        __device_add_disk+0x2ee [kernel]
        sd_probe_async+0xf5 [kernel]
        async_run_entry_fn+0x39 [kernel]
        process_one_work+0x1a7 [kernel]
        worker_thread+0x30 [kernel]
        kthread+0x112 [kernel]
        ret_from_fork+0x35 [kernel]

^C
```

Watch with eBPF what happens in /sys when a USB stick is inserted, with simple and complex examples.

In the first simple example above, the trace.py bcc tools script prints out a message whenever the sysfs_create_files() command runs. We see that sysfs_create_files() was started by a kworker thread in response to the USB stick insertion, but what file was created? The second example illustrates the full power of eBPF. Here trace.py is printing the kernel backtrace ("-K" option) plus name of the file created by sysfs_create_files(). The snippet inside the single quotes is some C source code, including an easily recognizable format string, that the provided Python script induces a LLVM just-in-time compiler to compile and then execute inside a in-kernel virtual machine. The full sysfs_create_files() function signature must be reproduced in the second command so that the format string can refer to one of the parameters. Making mistakes in this C snippet results in recognizable C-compiler errors. For example, if the "-I" parameter is omitted, the result is "Failed to compile BPF text." Developers who are conversant with either C or Python will find the bcc tools easy to extend and modify.

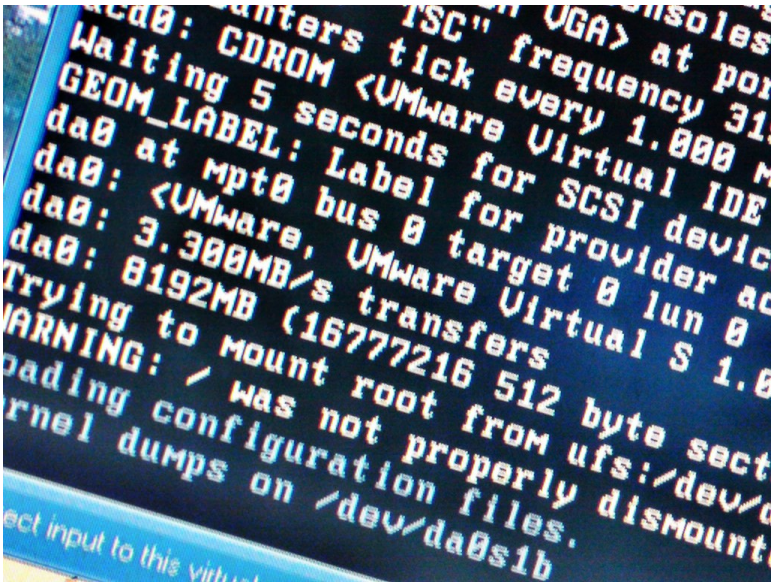
When the USB stick is inserted, the kernel backtrace appears showing that PID 7711 is a kworker thread that created a file called "events" in sysfs. A corresponding invocation with sysfs_remove_files() shows that removal of the USB stick results in removal of the events file, in keeping with the idea of reference-counting. Watching sysfs_create_link() with eBPF during USB stick insertion (not shown) reveals that no fewer than 48 symbolic links are created. The profusion of symlinks in /sys can be confirmed by typing "ls -R /sys".

What is the purpose of the "events" file anyway? Using cscope to find the function __device_add_disk() reveals that it calls disk_add_events() and that either "media_change" or "eject_request" may be written to the *events* file. Here the block layer of the kernel is informing

userspace about the appearance and disappearance of the "disk." Consider how quickly informative this method of investigating how USB stick insertion works is compared to trying to figure out the process solely from the source.

Read-only root filesystems make embedded devices possible

Assuredly no one shuts down a server or desktop system by pulling out the power plug. Why? Because mounted filesystems on the physical storage devices may have pending writes, and the data structures that record their state may be out of sync with what is actually written on the storage. When that happens, system owners will have to wait at next boot for the fsck filesystem-recovery tool to run, and in the worst case will actually lose data.



ro-rootfs are why embedded systems don't frequently need to fsck. Credit: https://tinyurl.com/yxoauoub

Yet, aficionados will have heard by now that many IoT and embedded devices like routers, thermostats and automobiles now run Linux. Many of these devices almost entirely lack a user interface, and there's no way to "unboot" them cleanly. Consider jump-starting a car with a dead battery, where the power to the Linux-running head unit goes up and down repeatedly. How is it that the system boots without a long fsck when the engine finally starts running? The answer is that embedded devices rely on a read-only root fileystem (ro-rootfs for short).

A ro-rootfs offers many advantages that are less obvious than incorruptibility. One is that malware cannot write to /usr or /lib if no Linux process can write there. Another is that a largely immutable filesystem is critical for field support of remote devices, as support personnel possess local systems that are nominally identical to those in the field. Perhaps the most important but also most subtle advantage is that ro-rootfs forces developers to decide during the *design phase* of a project which system objects will be immutable. Dealing with ro-rootfs may often be inconvenient or even painful, as "const variables" in programming languages often are, but the benefits easily repay the extra overhead.

Creating a read-only rootfs does require some additional amount of effort for embedded developers, and that's where VFS come in. Linux needs files in /var to be writable, and in addition, many popular applications that embedded systems run will try to create configuration dot-files in $HOME. One

solution for configuration files in the home directory is typically to pre-generate them and build them into the rootfs. For /var one approach is to mount it on a separate writable partition while / itself is mounted read-only. Using bind or overlay mounts is another popular alternative.

Bind and overlay mounts and their use by containers

"man mount" is the best place to learn about bind and overlay mounts, which give embedded developers and system administrators the power to create a filesystem in one path location and then provide it to applications at a second one. For embedded systems, the implication is that it's possible to store the files in /var on an unwritable flash device, but overlay- or bind-mount a path in a tmpfs onto the /var path at boot so that applications can scrawl there to their heart's delight. At next power-on, the changes in /var will be gone. Overlay mounts provide a union between the tmpfs and the underlying filesystem and allow apparent modification to an existing file in a ro-rootfs, while bind mounts can make new empty tmpfs directories show up as writable at ro-rootfs paths. While overlayfs is a proper filesystem type, bind mounts are implemented by the VFS namespace facility.

Based on the description of overlay and bind mounts, no one will be surprised that Linux containers make heavy use of them. Let's spy on what happens when we employ systemd-nspawn to start up a container by running bcc's mountsnoop tool:

```
second-shell$ sudo systemd-nspawn -D /srv/nspawn
Spawning container nspawn on /srv/nspawn.
Press ^] three times within 1s to kill container.
root@nspawn:~#
```

The system-nspawn invocation fires up the container while mountsnoop.py runs.

```
$ sudo ./mountsnoop.py
COMM             PID     TID     MNT_NS      CALL
systemd-nspawn   19374   19374   4026532702  mount("", "/", "", MS_NOSUID|MS_NOE
XEC|MS_REMOUNT|MS_BIND|MS_REC|MS_SILENT|MS_POSIXACL|MS_PRIVATE|MS_SLAVE|MS_SHARE
D|MS_RELATIME|MS_KERNMOUNT|MS_STRICTATIME|MS_LAZYTIME|MS_NOUSER|0x7fd430000300,
"") = 0
systemd-nspawn   19374   19374   4026532702  mount("/srv/nspawn", "/srv/nspawn",
 "", MS_NOSUID|MS_NOEXEC|MS_REMOUNT|MS_BIND|MS_REC|MS_SILENT|MS_POSIXACL|MS_PRIV
ATE|MS_SLAVE|MS_SHARED|MS_RELATIME|MS_KERNMOUNT|MS_STRICTATIME|MS_LAZYTIME|MS_NO
USER|0x7fd430000300, "") = 0
systemd-nspawn   19374   19374   4026532702  mount("", "/srv/nspawn", "", MS_NOS
UID|MS_NOEXEC|MS_REMOUNT|MS_BIND|MS_REC|MS_SILENT|MS_POSIXACL|MS_PRIVATE|MS_SLAV
E|MS_SHARED|MS_RELATIME|MS_KERNMOUNT|MS_STRICTATIME|MS_LAZYTIME|MS_NOUSER|0x7fd4
30000300, "") = 0
```

Running mountsnoop during the container "boot" reveals that the container runtime relies heavily on bind mounts. (Only the beginning of the lengthy output is displayed)

Here systemd-nspawn is providing selected files in the host's procfs and sysfs to the container at paths in its rootfs. Besides MS_BIND flag that sets bind-mounting, some of the other flags with which the "mount" system call was invoked determine what the relationship is between changes in the host namespace and that of the container.

Summary

Understanding Linux internals can seem an impossible task, as the kernel itself contains a gigantic amount of code, leaving aside Linux userspace applications and the system call interface in C libraries like glibc. One way to make progress is to read the source code of one kernel subsystem with an emphasis on understanding the userspace-facing system calls and headers plus major kernel internal interfaces, exemplified here by the file_operations table. The file operations are what makes "everything is a file" actually work, so getting a handle on them is particularly satisfying. The kernel C source files in the top-level fs/ directory constitute its implementation of virtual filesystems and are the shim layer that enables broad and relatively straightforward interoperability of popular storage-device filesystems and storage devices. Bind and overlay mounts via Linux namespaces are the VFS magic that make containers and read-only root filesystems possible. In combination with a study of source code, the eBPF kernel facility and its bcc interface makes probing the kernel simpler than ever before.

Acknowledgements

Much thanks to Akkana Peck and Michael Eager for comments and corrections.