Rethinking /dev and devices in the UNIX kernel

Poul-Henning Kamp
<phk@FreeBSD.org>
The FreeBSD Project

Abstract

An outstanding novelty in UNIX at its introduction was the notion of ‘‘a file is a file is a file and even a device is a file.’’ Going from ‘‘hardware only changes when the DEC Field engineer is here’’ to ‘‘my toaster has USB’’ has put serious strain on the rather crude implementation of the ‘‘devices as files’’ concept, an implementation which has survived practically unchanged for 30 years in most UNIX variants. Starting from a high-level view of devices and the semantics that have grown around them over the years, this paper takes the audience on a grand tour of the redesigned FreeBSD device-I/O system, to convey an overview of how it all fits together, and to explain why things ended up as they did, how to use the new features and in particular how not to.

1. Introduction

There are really only two fundamental ways to conceptualise I/O devices in an operating system: The usual way and the UNIX way.

The usual way is to treat I/O devices as their own class of things, possibly several classes of things, and provide APIs tailored to the semantics of the devices. In practice this means that a program must know what it is dealing with, it has to interact with disks one way, tapes another and rodents yet a third way, all of which are different from how it interacts with a plain disk file.

The UNIX way has never been described better than in the very first paper published on UNIX by Ritchie and Thompson [Ritchie74]:

> Special files constitute the most unusual feature of the UNIX filesystem. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory /dev, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file /dev/mt.
>
> Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special files are protected from indiscriminate access.
>
> There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

At the time, this was quite a strange concept; it was totally accepted for instance, that neither the system administrator nor the users were able to interact with a disk as a disk. Operating systems simply did not provide access to disk other than as a filesystem. Most vendors did not even release a program to initialise a disk-pack with a filesystem: selling pre-initialised and ‘‘quality tested’’ disk-packs was quite a profitable business.

In many cases some kind of API for reading and writing individual sectors on a disk pack did exist in the operating system, but more often than not it was not listed in the public documentation.

1.1. The traditional implementation

The initial implementation used hardcoded inode numbers [Ritchie98]. The console device would be inode number 5, the paper-tape-punch number 6 and so on, even if those inodes were also actual regular files in the filesystem.

For reasons one can only too vividly imagine, this was changed and Thompson [Thompson78] describes how the implementation now used ‘‘major and minor’’ device numbers to index though the devsw array to the correct device driver.

For all intents and purposes, this is the implementation which survives in most UNIX-like systems even to this day. Apart from the access control and timestamp information which is found in all inodes, the special inodes in the filesystem contain only one piece of information: the major and minor device numbers, often logically OR'ed to one field.

When a program opens a special file, the kernel uses the major number to find the entry points in the device driver, and passes the combined major and minor numbers as a parameter to the device driver.

2. The challenge

Now, we did not talk much about where the special inodes came from to begin with. They were created by hand, using the mknod(2) system call, usually through the mknod(8) program.

In those days a computer had a very static hardware configuration[1] and it certainly did not change while the system was up and running, so creating device nodes by hand was certainly an acceptable solution.

The first sign that this would not hold up as a solution came with the advent of TCP/IP and the telnet(1) program, or more precisely with the telnetd(8) daemon. In order to support remote login a ''pseudo-tty'' device driver was implemented, basically as tty driver which instead of hardware had another device which would allow a process to ''act as hardware'' for the tty. The telnetd(8) daemon would read and write data on the ''master'' side of the pseudo-tty and the user would be running on the ''slave'' side, which would act just like any other tty: you could change the erase character if you wanted to and all the signals and all that stuff worked.

Obviously with a device requiring no hardware, you can compile as many instances into the kernel as you like, as long as you do not use too much memory. As system after system was connected to the ARPANet, ''increasing number of ptys'' became a regular task for system administrators, and part of this task was to create more special nodes in the filesystem.

Several UNIX vendors also noticed an issue when they sold minicomputers in many different configurations: explaining to system administrators just which special nodes they would need and how to create them were a significant documentation hassle. Some opted for the simple solution and pre-populated /dev with every conceivable device node, resulting in a predictable slowdown on access to filenames in /dev.

[1] Unless your assigned field engineer was present on site.

System V UNIX provided a band-aid solution: a special boot sequence would take effect if the kernel or the hardware had changed since last reboot. This boot procedure would amongst other things create the necessary special files in the filesystem, based on an intricate system of per device driver configuration files.

In the recent years, we have become used to hardware which changes configuration at any time: people plug USB, Firewire and PCCard devices into their computers. These devices can be anything from modems and disks to GPS receivers and fingerprint authentication hardware. Suddenly maintaining the correct set of special devices in ''/dev'' became a major headache.

Along the way, UNIX kernels had learned to deal with multiple filesystem types [Heidemann91a] and a ''device-pseudo-filesystem'' was a pretty obvious idea. The device drivers have a pretty good idea which devices they have found in the configuration, so all that is needed is to present this information as a filesystem filled with just the right special files. Experience has shown that this like most other ''pseudo filesystems'' sound a lot simpler in theory than in practice.

3. Truly understanding devices

Before we continue, we need to fully understand the ''device special file'' in UNIX.

First we need to realize that a special file has the nature of a pointer from the filesystem into a different namespace; a little understood fact with far reaching consequences.

One implication of this is that several special files can exist in the filename namespace all pointing to the same device but each having their own access and timestamp attributes:

```
guest# ls -l /dev/fd0 /tmp/fd0
crw-r----- 1 root operator 9, 0 Sep 27 19:21 /dev/fd0
crw-rw-rw- 1 root wheel    9, 0 Sep 27 19:24 /tmp/fd0
```

Obviously, the administrator needs to be on top of this: one popular way to exploit an unguarded root prompt is to create a replica of the special file /dev/kmem in a location where it will not be noticed. Since /dev/kmem gives access to the kernel memory, gaining any particular privilege can be arranged by suitably modifying the kernel's data structures through the illicit special file.

When NFS appeared it opened a new avenue for this attack: People may have root privilege on one machine but not another. Since device nodes are not interpreted on the NFS server but rather on the local computer, a user with root privilege on a NFS client computer can create a device node to his liking on a filesystem mounted from an NFS server. This device node can in turn be used to circumvent the security of other computers which mount that filesystem, including the server, unless they protect themselves by not trusting any device entries on untrusted filesystem by mounting such filesystems with the `nodev` mount-option.

The fact that the device itself does not actually exist inside the filesystem which holds the special file makes it possible to perform boot-strapping stunts in the spirit of Baron Von Münchausen [raspe1785], where a filesystem is (re)mounted using one of its own device vnodes:

```
guest# mount -o ro /dev/fd0 /mnt
guest# fsck /mnt/dev/fd0
guest# mount -u -o rw /mnt/dev/fd0 /mnt
```

Other interesting details are chroot(2) and jail(2) [Kamp2000] which provide filesystem isolation for process-trees. Whereas chroot(2) was not implemented as a security tool [Mckusick1999] (although it has been widely used as such), the jail(2) security facility in FreeBSD provides a pretty convincing "virtual machine" where even the root privilege is isolated and restricted to the designated area of the machine. Obviously chroot(2) and jail(2) may require access to a well-defined subset of devices like /dev/null, /dev/zero and /dev/tty, whereas access to other devices such as /dev/kmem or any disks could be used to compromise the integrity of the jail(2) confinement.

For a long time FreeBSD, like almost all UNIX-like systems had two kinds of devices, "block" and "character" special files, the difference being that "block" devices would provide caching and alignment for disk device access. This was one of those minor architectural mistakes which took forever to correct.

The argument that block devices were a mistake is really very very simple: Many devices other than disks have multiple modes of access which you select by choosing which special file to use.

Pick any old timer and he will be able to recite painful sagas about the crucial difference between the /dev/rmt and /dev/nrmt devices for tape access.[2]

Tapes, asynchronous ports, line printer ports and many other devices have implemented submodes, selectable by the user at a special filename level, but that has not earned them their own special file types. Only disks[3] have enjoyed the privilege of getting an entire file type dedicated to a a minor device mode.

[2] Make absolutely sure you know the difference before you take important data on a multi-file 9-track tape to remote locations.

[3] Well, OK: and some 9-track tapes.

Caching and alignment modes should have been enabled by setting some bit in the minor device number on the disk special file, not by polluting the filesystem code with another file type.

In FreeBSD block devices were not even implemented in a fashion which would be of any use, since any write errors would never be reported to the writing process. For this reason, and since no applications were found to be in existence which relied on block devices and since historical usage was indeed historical [Mckusick2000], block devices were removed from the FreeBSD system. This greatly simlified the task of keeping track of open(2) reference counts for disks and removed much magic special-case code throughout.

4. Files, sockets, pipes, SVID IPC and devices

It is an instructive lesson in inconsistency to look at the various types of "things" a process can access in UNIX-like systems today.

First there are normal files, which are our reference yardstick here: they are accessed with open(2), read(2), write(2), mmap(2), close(2) and various other auxiliary system calls.

Sockets and pipes are also accessed via file handles but each has its own namespace. That means you cannot open(2) a socket,[4] but you can read(2) and write(2) to it. Sockets and pipes vector off at the file descriptor level and do not get in touch with the vnode based part of the kernel at all.

Devices land somewhere in the middle between pipes and sockets on one side and normal files on the other. They use the filesystem namespace, are implemented with vnodes, and can be operated on like normal files, but don't actually live in the filesystem.

Devices are in fact special-cased all the way through the vnode system. For one thing devices break the "one file-one vnode" rule, making it necessary to chain all vnodes for the same device together in order to be able to find "the canonical vnode for this device node", but more importantly, many operations have to be specifically denied on special file vnodes since they do not make any sense.

[4] This is particularly bizarre in the case of UNIX domain sockets which use the filesystem as their namespace and appear in directory listings.

For true inconsistency, consider the SVID IPC mechanisms - not only do they not operate via file handles, but they also sport a singularly illconceived 32 bit numeric namespace and a dedicated set of system calls for access.

Several people have convincingly argued that this is an inconsistent mess, and have proposed and implemented more consistent operating systems like the Plan9 from Bell Labs [Pike90a] [Pike92a]. Unfortunately reality is that people are not interested in learning a new operating system when the one they have is pretty darn good, and consequently research into better and more consistent ways is a pretty frustrating [Pike2000] but by no means irrelevant topic.

5. Solving the /dev maintenance problem

There are a number of obvious, simple but wrong ways one could go about solving the ‘‘/dev’’ maintenance problem.

The very straightforward way is to hack the namei() kernel function responsible for filename translation and lookup. It is only a minor matter of programming to add code to special-case any lookup which ends up in ‘‘/dev’’. But this leads to problems: in the case of chroot(2) or jail(2), the administrator will want to present only a subset of the available devices in ‘‘/dev’’, so some kind of state will have to be kept per chroot(2)/jail(2) about which devices are visible and which devices are hidden, but no obvious location for this information is available in the absence of a mount data structure.

It also leads to some unpleasant issues because of the fact that ‘‘/dev/foo’’ is a synthesised directory entry which may or may not actually be present on the filesystem which seems to provide ‘‘/dev’’. The vnodes either have to belong to a filesystem or they must be special-cased throughout the vnode layer of the kernel.

Finally there is the simple matter of generality: hard-coding the string "/dev" in the kernel is very general.

A cruder solution is to leave it to a daemon: make a special device driver, have a daemon read messages from it and create and destroy nodes in ‘‘/dev’’ in response to these messages.

The main drawback to this idea is that now we have added IPC to the mix introducing new and interesting race conditions.

Otherwise this solution is a surprisingly effective, but chroot(2)/jail(2) requirements prevents a simple implementation and running a daemon per jail would become an administrative nightmare.

Another pitfall of this approach is that we are not able to remount the root filesystem read-write at boot until we have a device node for the root device, but if this node is missing we cannot create it with a daemon since the root filesystem (and hence /dev) is read-only. Adding a read-write memory-filesystem mount /dev to solve this problem does not improve the architectural qualities further and certainly the KISS principle has been violated by now.

The final and in the end only satisfactory solution is to write a ‘‘DEVFS’’ which mounts on ‘‘/dev’’.

The good news is that it does solve the problem with chroot(2) and jail(2): just mount a DEVFS instance on the ‘‘dev’’ directory inside the filesystem subtree where the chroot or jail lives. Having a mountpoint gives us a convenient place to keep track of the local state of this DEVFS mount.

The bad news is that it takes a lot of cleanup and care to implement a DEVFS into a UNIX kernel.

6. DEVFS architectural decisions

Before implementing a DEVFS, it is necessary to decide on a range of corner cases in behaviour, and some of these choices have proved surprisingly hard to settle for the FreeBSD project.

6.1. The ‘‘persistence’’ issue

When DEVFS in FreeBSD was initially presented at a BoF at the 1995 USENIX Technical Conference in New Orleans, a group of people demanded that it provide ‘‘persistence’’ for administrative changes.

When trying to get a definition of ‘‘persistence’’, people can generally agree that if the administrator changes the access control bits of a device node, they want that mode to survive across reboots.

Once more tricky examples of the sort of manipulations one can do on special files are proposed, people rapidly disagree about what should be supported and what should not.

For instance, imagine a system with one floppy drive which appears in DEVFS as ‘‘/dev/fd0’’. Now the administrator, in order to get some badly written software to run, links this to ‘‘/dev/fd1’’:

```
ln /dev/fd0 /dev/fd1
```

This works as expected and with persistence in DEVFS, the link is still there after a reboot. But what if after a reboot another floppy drive has been connected to the system? This drive would naturally have the name "/dev/fd1", but this name is now occupied by the administrators hard link. Should the link be broken? Should the new floppy drive be called "/dev/fd2"? Nobody can agree on anything but the ugliness of the situation.

Given that we are no longer dependent on DEC Field engineers to change all four wheels to see which one is flat, the basic assumption that the machine has a constant hardware configuration is simply no longer true. The new assumption one should start from when analysing this issue is that when the system boots, we cannot know what devices we will find, and we can not know if the devices we do find are the same ones we had when the system was last shut down.

And in fact, this is very much the case with laptops today: if I attach my IOmega Zip drive to my laptop it appears like a SCSI disk named "/dev/da0", but so does the RAID-5 array attached to the PCI SCSI controller installed in my laptop's docking station. If I change mode to "a+rw" on the Zip drive, do I want that mode to apply to the RAID-5 as well? Unlikely.

And what if we have persistent information about the mode of device "/dev/sio0", but we boot and do not find any sio devices? Do we keep the information in our device-persistence registry? How long do we keep it? If I borrow a modem card, set the permissions to some non-standard value like 0666, and then attach some other serial device a year from now - do I want some old permissions changes to come back and haunt me, just because they both happened to be "/dev/sio0"? Unlikely.

The fact that more people have laptop computers today than five years ago, and the fact that nobody has been able to credibly propose where a persistent DEVFS would actually store the information about these things in the first place has settled the issue.

Persistence may be the right answer, but to the wrong question: persistence is not a desirable property for a DEVFS when the hardware configuration may change literally at any time.

6.2. Who decides on the names?

In a DEVFS-enabled system, the responsibility for creating nodes in /dev shifts to the device drivers, and consequently the device drivers get to choose the names of the device files. In addition an initial value for owner, group and mode bits are provided by the device driver.

But should it be possible to rename "/dev/lpt0" to "/dev/myprinter"? While the obvious affirmative answer is easy to arrive at, it leaves a lot to be desired once the implications are unmasked.

Most device drivers know their own name and use it purposefully in their debug and log messages to identify themselves. Furthermore, the "NewBus" [NewBus] infrastructure facility, which ties hardware to device drivers, identifies things by name and unit numbers.

A very common way to report errors in fact:

```
#define LPT_NAME "lpt" /* our official name */
[...]
printf(LPT_NAME
    ": cannot alloc ppbus (%d)!", error);
```

So despite the user renaming the device node pointing to the printer to "myprinter", this has absolutely no effect in the kernel and can be considered a userland aliasing operation.

The decision was therefore made that it should not be possible to rename device nodes since it would only lead to confusion and because the desired effect could be attained by giving the user the ability to create symlinks in DEVFS.

6.3. On-demand device creation

Pseudo-devices like pty, tun and bpf, but also some real devices, may not pre-emptively create entries for all possible device nodes. It would be a pointless waste of resources to always create 1000 ptys just in case they are needed, and in the worst case more than 1800 device nodes would be needed per physical disk to represent all possible slices and partitions.

For pseudo-devices the task at hand is to make a magic device node, "/dev/pty", which when opened will magically transmogrify into the first available pty subdevice, maybe "/dev/pty123".

Device submodes, on the other hand, work by having multiple entries in /dev, each with a different minor number, as a way to instruct the device driver in aspects of its operation. The most widespread example is probably "/dev/mt0" and "/dev/nmt0", where the node with the extra "n" instructs the tape device driver to not rewind on close.[5]

Some UNIX systems have solved the problem for pseudo-devices by creating magic cloning devices like "/dev/tcp". When a cloning device is opened, it finds a free instance and through vnode and file descriptor mangling return this new device to the opening process.

[5] This is the answer to the question in footnote number 2.

This scheme has two disadvantages: the complexity of switching vnodes in midstream is non-trivial, but even worse is the fact that it does not work for submodes for a device because it only reacts to one particular /dev entry.

The solution for both needs is a more flexible on-demand device creation, implemented in FreeBSD as a two-level lookup. When a filename is looked up in DEVFS, a match in the existing device nodes is sought first and if found, returned. If no match is found, device drivers are polled in turn to ask if they would be able to synthesise a device node of the given name.

The device driver gets a chance to modify the name and create a device with make_dev(). If one of the drivers succeeds in this, the lookup is started over and the newly found device node is returned:

```
pty_clone()
   if (name != "pty")
      return(NULL); /* no luck */
   n = find_next_unit();
   dev = make_dev(...,n,"pty%d",n);
   name = dev->name;
   return(dev);
```

An interesting mixed use of this mechanism is with the sound device drivers. Modern sound devices have multiple channels, presumably to allow the user to listen to CNN, Napstered MP3 files and Quake sound effects at the same time. The only problem is that all applications attempt to open ‘‘/dev/dsp’’ since they have no concept of multiple sound devices. The sound device drivers use the cloning facility to direct ‘‘/dev/dsp’’ to the first available sound channel completely transparently to the process.

There are very few drawbacks to this mechanism, the major one being that ‘‘ls /dev’’ now errs on the sparse side instead of the rich when used as a system device inventory, a practice which has always been of dubious precision at best.

6.4. Deleting and recreating devices

Deleting device nodes is no problem to implement, but as likely as not, some people will want a method to get them back. Since only the device driver know how to create a given device, recreation cannot be performed solely on the basis of the parameters provided by a process in userland.

In order to not complicate the code which updates the directory structure for a mountpoint to reflect changes in the DEVFS inode list, a deleted entry is merely marked with DE_WHITEOUT instead of being removed entirely. Otherwise a separate list would be needed for inodes which we had deleted so that they would not be mistaken for new inodes.

The obvious way to recreate deleted devices is to let mknod(2) do it by matching the name and disregarding the major/minor arguments. Recreating the device with mknod(2) will simply remove the DE_WHITEOUT flag.

6.5. Jail(2), chroot(2) and DEVFS

The primary requirement from facilities like jail(2) and chroot(2) is that it must be possible to control the contents of a DEVFS mount point.

Obviously, it would not be desirable for dynamic devices to pop into existence in the carefully pruned /dev of jails so it must be possible to mark a DEVFS mountpoint as ‘‘no new devices’’. And in the same way, the jailed root should not be able to recreate device nodes which the real root has removed.

These behaviours will be controlled with mount options, but these have not yet been implemented because FreeBSD has run out of bitmap flags for mount options, and a new unlimited mount option implementation is still not in place at the time of writing.

One mount option ‘‘jaildevfs’’, will restrict the contents of the DEVFS mountpoint to the ‘‘normal set’’ of devices for a jail and automatically hide all future devices and make it impossible for a jailed root to un-hide hidden entries while letting an un-jailed root do so.

Mounting or remounting read-only, will prevent all future devices from appearing and will make it impossible to hide or un-hide entries in the mountpoint. This is probably only useful for chroots or jails where no tty access is intended since cloning will not work either.

More mount options may be needed as more experience is gained.

6.6. Default mode, owner & group

When a device driver creates a device node, and a DEVFS mount adds it to its directory tree, it needs to have some values for the access control fields: mode, owner and group.

Currently, the device driver specifies the initial values in the make_dev() call, but this is far from optimal. For one thing, embedding magic UIDs and GIDs in the kernel is simply bad style unless they are numerically zero. More seriously, they represent compile-time defaults which in these enlightened days is rather old-fashioned.

7. Cleaning up before we build: struct specinfo and dev_t

Most of the rest of the paper will be about the various challenges and issues in the implementation of DEVFS in FreeBSD. All of this should be applicable to other systems derived from 4.4BSD-Lite as well.

POSIX has defined a type called ‘‘dev_t’’ which is the identity of a device. This is mainly for use in the few system calls which knows about devices: stat(2), fstat(2) and mknod(2). A dev_t is constructed by logically OR'ing the major# and minor# for the device. Since those have been defined as having no overlapping bits, the major# and minor# can be retrieved from the dev_t by a simple masking operation.

Although the kernel had a well-defined concept of any particular device it did not have a data structure to represent "a device". The device driver has such a structure, traditionally called ‘‘softc’’ but the high kernel does not (and should not!) have access to the device driver's private data structures.

It is an interesting tale how things got to be this way,[6] but for now just record for a fact how the actual relationship between the data structures was in the 4.4BSD release (Fig. 1). [44BSDBook]

As for all other files, a vnode references a filesystem inode, but in addition it points to a ‘‘specinfo’’ structure. In the inode we find the dev_t which is used to reference the device driver.

Access to the device driver happens by extracting the major# from the dev_t, indexing through the global devsw[] array to locate the device driver's entry point.

The device driver will extract the minor# from the dev_t and use that as the index into the softc array of private data per device.

The ‘‘specinfo’’ structure is a little sidekick vnodes grew underway, and is used to find all vnodes which reference the same device (i.e. they have the same major# and minor#). This linkage is used to determine which vnode is the ‘‘chosen one’’ for this device, and to keep track of open(2)/close(2) against this device. The actual implementation was an inefficient hash implementation, which depending on the vnode reclamation rate and /dev directory lookup traffic, may become a measurable performance liability.

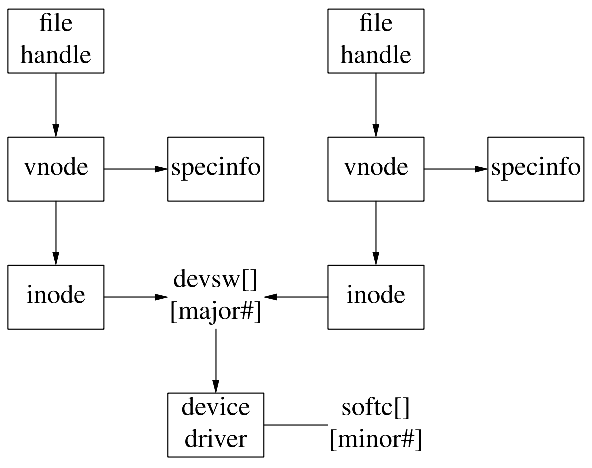

Fig. 1 - Data structures in 4.4BSD

[6] Basically, devices should have been moved up with sockets and pipes at the file descriptor level when the VFS layering was introduced, rather than have all the special casing throughout the vnode system.

7.1. The new vnode/inode/dev_t layout

In the new layout (Fig. 2) the specinfo structure takes a central role. There is only one instanace of struct specinfo per device (i.e. unique major# and minor# combination) and all vnodes referencing this device point to this structure directly.

In userland, a dev_t is still the logical OR of the major# and minor#, but this entity is now called a udev_t in the kernel. In the kernel a dev_t is now a pointer to a struct specinfo.

All vnodes referencing a device are linked to a list hanging directly off the specinfo structure, removing the need for the hash table and consequently simplifying and speeding up a lot of code dealing with vnode instantiation, retirement and name-caching.

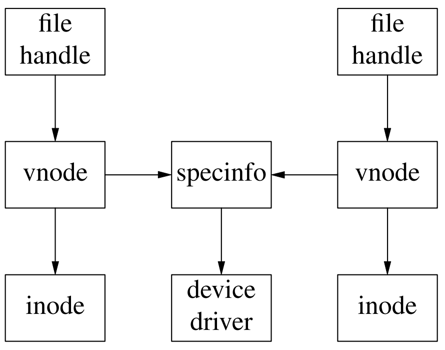

Fig. 2 - The new FreeBSD data structures.

The entry points to the device driver are stored in the specinfo structure, removing the need for the devsw[] array and allowing device drivers to use separate entry-points for various minor numbers.

This is very convenient for devices which have a “control” device for management and tuning. The control device, almost always have entirely separate open/close/ioctl implementations [MD.C].

In addition to this, two data elements are included in the specinfo structure but “owned” by the device driver. Typically the device driver will store a pointer to the softc structure in one of these, and unit number or mode information in the other.

This removes the need for drivers to find the softc using array indexing based on the minor#, and at the same time has obliviated the need for the compiled-in “NFOO” constants which traditionally determined how many softc structures and therefore devices the driver could support.[7]

There are some trivial technical issues relating to allocating the storage for specinfo early in the boot sequence and how to find a specinfo from the udev_t/major#+minor#, but they will not be discussed here.

7.2. Creating and destroying devices

Ideally, devices should only be created and destroyed by the device drivers which know what devices are present. This is accomplished with the make_dev() and destroy_dev() function calls.

Life is seldom quite that simple. The operating system might be called on to act as a NFS server for a diskless workstation, possibly even of a different architecture, so we still need to be able to represent device nodes with no device driver backing in the filesystems and consequently we need to be able to create a specinfo from the major#+minor# in these inodes when we encounter them. In practice this is quite trivial, but in a few places in the code one needs to be aware of the existence of both “named” and “anonymous” specinfo structures.

The make_dev() call creates a specinfo structure and populates it with driver entry points, major#, minor#, device node name (for instance “lpt0”), UID, GID and access mode bits. The return value is a dev_t (i.e., a pointer to struct specinfo). If the device driver determines that the device is no longer present, it calls destroy_dev(), giving a dev_t as argument and the dev_t will be cleaned and converted to an anonymous dev_t.

[7] Not to mention all the drivers which implemented panic(2) because they forgot to perform bounds checking on the index before using it on their softc arrays.

Once created with make_dev() a named dev_t exists until destroy_dev() is called by the driver. The driver can rely on this and keep state in the fields in dev_t which is reserved for driver use.

8. DEVFS

By now we have all the relevant information about each device node collected in struct specinfo but we still have one problem to solve before we can add the DEVFS filesystem on top of it.

8.1. The interrupt problem

Some device drivers, notably the CAM/SCSI subsystem in FreeBSD will discover changes in the device configuration inside an interrupt routine.

This imposes some limitations on what can and should do be done: first one should minimise the amount of work done in an interrupt routine for performance reasons; second, to avoid deadlocks, vnodes and mount-points should not be accessed from an interrupt routine.

Also, in addition to the locking issue, a machine can have many instances of DEVFS mounted: for a jail(8) based virtual-machine web-server several hundred instances is not unheard of, making it far too expensive to update all of them in an interrupt routine.

The solution to this problem is to do all the filesystem work on the filesystem side of DEVFS and use atomically manipulated integer indices (“inode numbers”) as the barrier between the two sides.

The functions called from the device drivers, make_dev(), destroy_dev() &c. only manipulate the DEVFS inode number of the dev_t in question and do not even get near any mountpoints or vnodes.

For make_dev() the task is to assign a unique inode number to the dev_t and store the dev_t in the DEVFS-global inode-to-dev_t array.

```
make_dev(...)
    store argument values in dev_t
    assign unique inode number to dev_t
    atomically insert dev_t into inode_array
```

For destroy_dev() the task is the opposite: clear the inode number in the dev_t and NULL the pointer in the devfs-global inode-to-dev_t array.

```
destroy_dev(...)
    clear fields in dev_t
    zero dev_t inode number.
    atomically clear entry in inode_array
```

Both functions conclude by atomically incrementing a global variable `devfs_generation` to leave an indication to the filesystem side that something has changed.

By modifying the global state only with atomic instructions, locks have been entirely avoided in this part of the code which means that the make_dev() and destroy_dev() functions can be called from practically anywhere in the kernel at any time.

On the filesystem side of DEVFS, the only two vnode methods which examine or rely on the directory structure, VOP_LOOKUP and VOP_READDIR, call the function devfs_populate() to update their mountpoint's view of the device hierarchy to match current reality before doing any work.

```
devfs_readdir(...)
    devfs_populate(...)
    ...
```

The devfs_populate() function, compares the current `devfs_generation` to the value saved in the mountpoint last time devfs_populate() completed and if (actually: while) they differ a linear run is made through the devfs-global inode-array and the directory tree of the mountpoint is brought up to date.

The actual code is slightly more complicated than shown in the pseudo-code here because it has to deal with subdirectories and hidden entries.

```
devfs_populate(...)
  while (mount->generation != devfs_generation)
    for i in all inodes
      if inode created)
        create directory entry
      else if inode destroyed
        remove directory entry
```

Access to the global DEVFS inode table is again implemented with atomic instructions and failsafe retries to avoid the need for locking.

From a performance point of view this scheme also means that a particular DEVFS mountpoint is not updated until it needs to be, and then always by a process belonging to the jail in question thus minimising and distributing the CPU load.

9. Device-driver impact

All these changes have had a significant impact on how device drivers interact with the rest of the kernel regarding registration of devices.

If we look first at the "before" image in Fig. 3, we notice first the NFOO define which imposes a firm upper limit on the number of devices the kernel can deal with. Also notice that the softc structure for all of them is allocated at compile time. This is because most device drivers (and texts on writing device drivers) are from before the general kernel malloc facility [Mckusick1988] was introduced into the BSD kernel.

```
#ifndef NFOO
#       define NFOO    4
#endif

struct foo_softc {
        ...
} foo_softc[NFOO];

int nfoo = 0;

foo_open(dev, ...)
{
        int unit = minor(dev);
        struct foo_softc *sc;

        if (unit >= NFOO || unit >= nfoo)
                return (ENXIO);

        sc = &foo_softc[unit]

        ...
}

foo_attach(...)
{
        struct foo_softc *sc;
        static int once;

        ...
        if (nfoo >= NFOO) {
                /* Have hardware, can't handle */
                return (-1);
        }
        sc = &foo_softc[nfoo++];
        if (!once) {
                cdevsw_add(&cdevsw);
                once++;
        }
        ...
}
```

Fig. 3 - Device-driver, old style.

Also notice how range checking is needed to make sure that the minor# is inside range. This code gets more complex if device-numbering is sparse. Code equivalent to that shown in the foo_open() routine would also be needed in foo_read(), foo_write(), foo_ioctl() &c.

Finally notice how the attach routine needs to remember to register the cdevsw structure (not shown) when the first device is found.

Now, compare this to our "after" image in Fig. 4. NFOO is totally gone and so is the compile time allocation of space for softc structures.

The foo_open (and foo_close, foo_ioctl &c) functions can now derive the softc pointer directly from the dev_t they receive as an argument.

```
struct foo_softc {
        ....
};

int nfoo;

foo_open(dev, ...)
{
        struct foo_softc *sc = dev->si_drv1;

        ...
}

foo_attach(...)
{
        struct foo_softc *sc;

        ...
        sc = MALLOC(..., M_ZERO);
        if (sc == NULL) {
                /* Have hardware, can't handle */
                return (-1);
        }
        sc->dev = make_dev(&cdevsw, nfoo,
            UID_ROOT, GID_WHEEL, 0644,
            "foo%d", nfoo);
        nfoo++;
        sc->dev->si_drv1 = sc;
        ...
}
```

Fig. 4 - Device-driver, new style.

In foo_attach() we can now attach to all the devices we can allocate memory for and we register the cdevsw structure per dev_t rather than globally.

This last trick is what allows us to discard all bounds checking in the foo_open() &c. routines, because they can only be called through the cdevsw, and the cdevsw is only attached to dev_t's which foo_attach() has created. There is no way to end up in foo_open() with a dev_t not created by foo_attach().

In the two examples here, the difference is only 10 lines of source code, primarily because only one of the worker functions of the device driver is shown. In real device drivers it is not uncommon to save 50 or more lines of source code which typically is about a percent or two of the entire driver.

10. Future work

Apart from some minor issues to be cleaned up, DEVFS is now a reality and future work therefore is likely concentrate on applying the facilities and functionality of DEVFS to FreeBSD.

10.1. devd

It would be logical to complement DEVFS with a ‘‘device-daemon’’ which could configure and de-configure devices as they come and go. When a disk appears, mount it. When a network interface appears, configure it. And in some configurable way allow the user to customise the action, so that for instance images will automatically be copied off the flash-based media from a camera, &c.

In this context it is good to question how we view dynamic devices. If for instance a printer is removed in the middle of a print job and another printer arrives a moment later, should the system automatically continue the print job on this new printer? When a disk-like device arrives, should we always mount it? Should we have a database of known disk-like devices to tell us where to mount it, what permissions to give the mount-point? Some computers come in multiple configurations, for instance laptops with and without their docking station. How do we want to present this to the users and what behaviour do the users expect?

10.2. Pathname length limitations

In order to simplify memory management in the early stages of boot, the pathname relative to the mountpoint is presently stored in a small fixed size buffer inside struct specinfo. It should be possible to use filenames as long as the system otherwise permits, so some kind of extension mechanism is called for.

Since it cannot be guaranteed that memory can be allocated in all the possible scenarios where make_dev() can be called, it may be necessary to mandate that the caller allocates the buffer if the content will not fit inside the default buffer size.

10.3. Initial access parameter selection

As it is now, device drivers propose the initial mode, owner and group for the device nodes, but it would be more flexible if it were possible to give the kernel a set of rules, much like packet filtering rules, which allow the user to set the wanted policy for new devices. Such a mechanism could also be used to filter new devices for mount points in jails and to determine other behaviour.

Doing these things from userland results in some awkward race conditions and software bloat for embedded systems, so a kernel approach may be more suitable.

10.4. Applications of on-demand device creation

The facility for on-demand creation of devices has some very interesting possibilities.

One planned use is to enable user-controlled labelling of disks. Today disks have names like /dev/da0, /dev/ad4, but since this numbering is topological any change in the hardware configuration may rename the disks, causing /etc/fstab and backup procedures to get out of sync with the hardware.

The current idea is to store on the media of the disk a user-chosen disk name and allow access through this name, so that for instance /dev/mydisk0 would be a symlink to whatever topological name the disk might have at any given time.

To simplify this and to avoid a forest of symlinks, it will probably be decided to move all the sub-divisions of a disk into one subdirectory per disk so just a single symlink can do the job. In practice that means that the current /dev/ad0s2f will become something like /dev/ad0/s2f and so on. Obviously, in the same way, disks could also be accessed by their topological address, down to the specific path in a SAN environment.

Another potential use could be for automated offline data media libraries. It would be quite trivial to make it possible to access all the media in the library using /dev/lib/$LABEL which would be a remarkable simplification compared with most current automated retrieval facilities.

Another use could be to access devices by parameter rather than by name. One could imagine sending a printjob to /dev/printer/color/A2 and behind the scenes a search would be made for a device with the correct properties and paper-handling facilities.

11. Conclusion

DEVFS has been successfully implemented in FreeBSD, including a powerful, simple and flexible solution supporting pseudo-devices and on-demand device node creation.

Contrary to the trend, the implementation added functionality with a net decrease in source lines, primarily because of the improved API seen from device drivers point of view.

Even if DEVFS is not desired, other 4.4BSD derived UNIX variants would probably benefit from adopting the dev_t/specinfo related cleanup.

12. Acknowledgements

I first got started on DEVFS in 1989 because the abysmal performance of the Olivetti M250 computer forced me to implement a network-disk-device for Minix in order to retain my sanity. That initial work led to a crude but working DEVFS for Minix, so obviously both Andrew Tannenbaum and Olivetti deserve credit for inspiration.

Julian Elischer implemented a DEVFS for FreeBSD around 1994 which never quite made it to maturity and subsequently was abandoned.

Bruce Evans deserves special credit not only for his keen eye for detail, and his competent criticism but also for his enthusiastic resistance to the very concept of DEVFS.

Many thanks to the people who took time to help me stamp out "Danglish" through their reviews and comments: Chris Demetriou, Paul Richards, Brian Somers, Nik Clayton, and Hanne Munkholm. Any remaining insults to proper use of english language are my own fault.

13. References

[44BSDBook] Mckusick, Bostic, Karels & Quarterman: "The Design and Implementation of 4.4 BSD Operating System." Addison Wesley, 1996, ISBN 0-201-54979-4.

[Heidemann91a] John S. Heidemann: "Stackable layers: an architecture for filesystem development." Master's thesis, University of California, Los Angeles, July 1991. Available as UCLA technical report CSD-910056.

[Kamp2000] Poul-Henning Kamp and Robert N. M. Watson: "Confining the Omnipotent root." Proceedings of the SANE 2000 Conference. Available in FreeBSD distributions in `/usr/share/papers`.

[MD.C] Poul-Henning Kamp et al: FreeBSD memory disk driver: `src/sys/dev/md/md.c`

[Mckusick1988] Marshall Kirk Mckusick, Mike J. Karels: "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX-Kernel" Proceedings of the San Francisco USENIX Conference, pp. 295-303, June 1988.

[Mckusick1999] Dr. Marshall Kirk Mckusick: Private email communication. *"According to the SCCS logs, the chroot call was added by Bill Joy on March 18, 1982 approximately 1.5 years before 4.2BSD was released. That was well before we had ftp servers of any sort (ftp did not show up in the source tree until January 1983). My best guess as to its purpose was to* *allow Bill to chroot into the /4.2BSD build directory and build a system using only the files, include files, etc contained in that tree. That was the only use of chroot that I remember from the early days."*

[Mckusick2000] Dr. Marshall Kirk Mckusick: Private communication at BSDcon2000 conference. *"I have not used block devices since I wrote FFS and that was* many *years ago."*

[NewBus] NewBus is a subsystem which provides most of the glue between hardware and device drivers. Despite the importance of this there has never been published any good overview documentation for it. The following article by Alexander Langer in "Dæmonnews" is the best reference I can come up with: `http://www.daemonnews.org/200007/newbus-intro.html`

[Pike2000] Rob Pike: "Systems Software Research is Irrelevant."
`http://www.cs.bell-labs.com/who/rob/utah2000.pdf`

[Pike90a] Rob Pike, Dave Presotto, Ken Thompson and Howard Trickey: "Plan 9 from Bell Labs." Proceedings of the Summer 1990 UKUUG Conference.

[Pike92a] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey and Phil Winterbottom: "The Use of Name Spaces in Plan 9." Proceedings of the 5th ACM SIGOPS Workshop.

[Raspe1785] Rudolf Erich Raspe: "Baron Münchhausen's Narrative of his marvellous Travels and Campaigns in Russia." Kearsley, 1785.

[Ritchie74] D.M. Ritchie and K. Thompson: "The UNIX Time-Sharing System" Communications of the ACM, Vol. 17, No. 7, July 1974.

[Ritchie98] Dennis Ritchie: private conversation at USENIX Annual Technical Conference New Orleans, 1998.

[Thompson78] Ken Thompson: "UNIX Implementation" The Bell System Technical Journal, vol 57, 1978, number 6 (part 2) p. 1931ff.