Thoughts on a Possible 64-bit DOS OS

krailethh

January 23, 2026

1 A Little Rant (to get in the Mood)

I'm a die-hard Unix lover. But to keep yourself honest, you must not refrain from relentlessly criticising what you love. So let me start with a little rant that's very much opinionated and totally does not aim at being a balanced analysis.

The very idea of Unix was that of simplicity (over the complexity of Multics). Yet, if we look at a typical modern Unix-like, we see some of the most complex beasts that are out there. Some are a bit leaner (like NetBSD), others are so ridiculously oversized (the wider GNU/Linux ecosystem) and as much of a self-parody that people had to build an editor on top of a Web browser engine (Atom) to beat the level of absurdity. Why is this? There are two reasons for it:

1. Historic baggage: Unix is weight down by 50+ years worth of careful evolution driven by the desire to not break things and retain compatibility. For example, physical *terminals* died out so long ago that two entire social generations have been born after their demise. And here I am today, typing this text on my keyboard, causing signals which the event input driver of X11 receives – only to feed the data to a *terminal emulator* application! All that just to pretend everything is still like it was in 1970, when Thompson and Ritchie bootstrapped Unix on Bell's PDP-7 minicomputer.

   Another example? How about us using filesystems that still insist on 'ending on a cylinder boundary' even though most of our storage media don't spin anymore (I hope!) and have no platters to begin with? In far too many ways we're walled in by such anachronisms.

2. General purpose aspiration: While the power of Unix comes from its components adhering to the idea of 'doing one thing and doing it well', the trend shifted to attempting to do everything – and doing a lousy job at that (I don't even have to name the biggest offender here, do I?).

   And as if trying to do absolutely everything wasn't bad enough, people seem to enjoy solving (relatively) simple problems with terribly complex means. So much so that eventually you need an abstraction layer for the abstraction layer, because it's too hard to use! A dangerous mindset has been running wild for quite a while now: 'Fix that stupid piece of software? Nah, let's stuff it into a "pod" and build an orchestration engine that can keep n instances of it running all the time, then it's no problem if it crashes once a week.'

Of course the future looks bright. How couldn't it when so many people choose to describe every problem to Claude rather than typing `man $SOMETHING?` Need I say more? Probably not – either you understand my exasperation by now or we simply disagree on the matter.

2 The DOS OS Family

So what's the solution to this problem? Keep reinventing the `gid 0` over and over again? Yes and no. Not necessarily all the time – never reaching production quality with anything is not exactly helpful. But there's a lot of value in all kinds of experimentation and just trying out things. To this day, people like to study

Plan 9 for example, even though probably nobody expects it to ever "succeed" on the market. Which is totally fine for a research OS!

Recently, over on r/kerneldevelopment, a project idea was posted that basically boils down to a *64-bit DOS system* for modern hardware. I like both the idea and the spirit ('let's do it for fun and see what happens').

The DOS family of operating systems is actually a pretty fascinating one, and while MS-DOS is by far the best known one, there are others that are vastly more interesting. It all began with the 8-bit microcomputer OS `CP/M`, first released in 1974 by Digital Research. It's major innovation was that it splits the core OS into two components, the BIOS (Basic Input/Output System) and the BDOS (Basic Disk Operating System). While the former was hardware-dependent, the latter used routines provided by the BIOS. This makes the system much more portable since only the BIOS portion needed to be adapted to various platforms and the BDOS would just work.

How influential this model has become is obvious: in 1981, for its PC architecture, IBM introduced a system BIOS or ROM BIOS on which operating systems could rely. IBM had tricked Digital Research, and the PC was overwhelmingly sold with DOS—at the time a cheap CP/M rip-off. Due to the success of the PC architecture, DOS eventually displaced CP/M.

There are various important and well-known innovations that originate within MS-DOS like the hierarchical FAT filesystem that allowed for subdirectories (CP/M had a flat filesystem). Some of the features of CP/M that didn't make it into DOS or from other DOS systems beside MS-DOS are not widely known today, though. For example, while CP/M didn't support subdirectories, it came with a facility known as *user areas* which allowed for organizing files. Then there was `MP/M`, first released in 1979, which was a multi-user and multi-tasking variant of CP/M. In 1982, Digital Research released `CCP/M`, a single-user multi-tasking system that introduced virtual screens.

The MS-DOS line of operating systems evolved into a program starter for Windows 9.x and eventually into a legacy layer within Windows NT. Microsoft were a major Unix vendor during the DOS era and had no interest in adding advanced features to MS-DOS so it wouldn't compete with Xenix. The same was not true for Digital Research, who came up with amazing innovations time and time again. Several underhanded manoeuvrers by Microsoft as well as a couple of bad decisions on Digital Research's part prevented their OS family ever taking the lead again. The last offspring of their line, `REAL/32`, was discontinued in the late 2000s.

So what kind of value could a DOS64 operating system actually provide other than being highly educational? A lot, surprisingly. At its core it would probably be a platform used by retro enthusiasts and I wouldn't be surprised if people ported games to it or wrote new ones. If network functionality was to be added, it would make a fun server OS for the BBS scene. And it might be a great exercise in minimalism for people who'd like to explore the how far they can take *simple*.

3 Platform and Toolchain Considerations

Let's imagine DOS/64 (or, if that name sounds too unimaginative, how about "Exhumed OS", which, tongue-in-cheek, could be abbreviated "OS Ex"?)!

The suggestion was to write it in C, following the modern C23 standard. If self-hosting is clearly a non-goal, that is great. If we want to consider self-hosting eventually, though, C standards should be re-considered. TCC supports ANSI C and 'much of the C99 standard'. PCC is C99. SCC (a full toolchain including cc, cpp, make, as, ld, ar and more) also targets C99. Finally CPROC is C11 compiler which already supports some C23 features.

Another crucial decision to make is the platform. Will it be targeting *amd64* only (as that's the HW people usually have to play with) or are *arm64* and *riscv64* also a goal? If the former, the project could use the EFI for its boot loader and avoid the pitfalls of `uboot` related stuff.

In general, beginning a new OS from scratch offers a unique chance: starting legacy-free without worrying about all the old cruft that others have to deal with. Obviously to jus-

tify a name like "DOS/64", the system would have to be recognizable as a DOS. Of course this is mostly about the user-facing portion of the OS and the internals can be (and probably should be) vastly different from how 16-bit DOS or even FreeDOS works. A decision needs to be made if the system should be a *DOS-like* (which would remain pretty close to what DOS felt like) or *DOS-inspired* (which could deviate further from its original role model). Both options have their own appeal: the former would be 'DOS, but 64-bit', while the latter would explore what DOS might have evolved to. I think option 2 would be more interesting as it can develop organically while the other one necessarily would feel like an anachronism which only caters to the retro feeling.

What makes DOS recognizable are things like the `COMMAND.COM` command interpreter and the 8.3 filename schema. I think it makes sense to implement rw support for FAT32 and stick with that rather than trying to design a new FS. If we want more modern features, it might make sense to consider a layered storage framework like FreeBSD's GEOM (which is the most flexible solution I know of, basically *storage done right*). GEOM offers various classes; instances of them are *consumers* of underlying *providers*, transform them and become a provider themselves.

For example the *gmirror* class consumes two storage providers (partitions, physical disks, memdisks, other geom providers) and acts as a new provider for a RAID1-type storage. You can have *gjournal* consume it and become another provider. If you create a FAT32 FS on top of the latter, you have a journalled, softraid FAT32 that you could pass to a DOS VM and the OS would be perfectly happy to work with it! Seriously: how cool is that?

You also use classes for things like encryption or even remote storage (*geom gate*) and there also was a GEOM-based volume manager (*gvinum*). I think taking some inspiration from that and solve some problems at different layers than the actual filesystem might make sense – sticking with trusty and compatible FAT32, but giving people who want some more the option to implement components for a storage stack.

4 Storage Devices and Access

Bringing DOS-style handling of drives into the modern age is a bit of a challenge. Back in the day you either had fixed disks or drives with ejectable media – but devices couldn't connect or vanish during runtime. Thanks to USB, they now can. Also DOS used to read and write filesystems *hot*. That was terrible but 'good enough' practice even then. But purely synchronous writes are an anachronism, and if we want to buffer writes, we need to lock and release filesystems. Unix' concept of *mounting* is not idiomatic to DOS and doesn't feel like a good candidate to bring over for multiple reasons:

- in DOS, we organize files in several device-based trees, the root of which is a drive letter. There is no global hierarchy where an additional filesystem can be mounted to just about any node.
- we don't have a device subtree (/dev/$something), either.

Traditionally, DOS uses special names to refer to devices like `CON` for the console. This is extremely ugly and can bite people (to this day you cannot create a file named `CON`, `COM1`, `LPT3` and so on in Windows!). It might be better for the kernel to keep a list of possible device names in a special namespace which separates them from filenames. Since USB supports *hubs* and can become rather complex, let's for example reserve 10 USB device nodes: USB1 .. USB10. Most commands never need to access those. For those that do, we can use a special syntax of starting with a backslash (to denote an absolute path) but without a drive letter (thus pointing to the device list provided by the kernel). So for example, 'USB1' would be a valid filename without an extension in the local directory that can be accessed with a relative path (`USB1`) or with an absolute path (e.g. `C:\SOMEWHERE\MYFILES\USB1`). Referring to the device 'USB1' would be done like this: `\USB1.`

Running `DEVICE /R` returns all the reserved names in the device tree. `DEVICE /C \USB11` would create an additional device node. `DEVICE /D \USB10` would error out, as device

names need to be sequential and USB10 cannot be deleted when USB11 exists. Running just `DEVICE` without any parameters displays which ones are actually connected to a device and the device info if available (like 'Acme 16GB USB thumb drive') which the kernel might have exposed as USB1. If the USB drive contains two partitions with usable filesystems, they might be exposed as USB1 and USB2.

Knowing that the storage device we want to access is called USB1, we could then issue the command `ASSIGN \USB1 E:`. This would lock the filesystem and make it available to common programs under the drive letter E. When the user is done with it, he needs to issue `UNASSIGN E:` to release it. The command blocks until any cache buffers are flushed, and then reports that the drive is safe to remove (unless other partitions of the same device are still in use?).

A variant of the device naming would be that the kernel doesn't simply pick the next free name for every partition but does something slightly more sophisticated. Here's the catch with the simple schema proposed so far: Imagine an unpartitioned USB stick with a FAT32 FS getting attached to the system. The kernel registers \USB1 for it. Then a second one just like it is attached. It gets registered as \USB2. New we remove the first one; \USB1 is now a reserved (unattached) node. When we now connect another USB stick, one that has two FAT32 partitions, the kernel would expose the first one as \USB1 and the second one as \USB3. Them not being sequential could be confusing.

To solve this, we can either have the kernel find two continuous ones (thus ignoring \USB1 in this case), or we could solve the problem by thinking of the device node as the whole device and use a name + extension notation to refer to the partitions – \USB1.1 and \USB1.2 in this case. This would make it easier for the `UNASSIGN` command to figure out if any other subdevices of the same device are still assigned and print a warning rather than 'it's safe to remove'. The former case is simpler, the latter more elegant and more useful (and covers cases like `FDISK \USB1` when a program needs to access the device to change partitioning!). However with the more sophisticated one the question is if the subnodes need to be created with `DEVICE /C \USB1.1`, too, or if this pseudo-filesystem tree is complex enough to warrant a facility for dynamically creating and destroying nodes when devices attach or vanish (devd / udev like).

Maybe it also makes sense to make the backslash special namespace hierarchical? Then the drive nodes would be specified like this: \DEV\USB1. This would allow using it for other things in the future, too, for example net shares like \NET\SHARE1.