Live blog day 1 — morning

by Anisse ASTIER | Sep 25, 2019 | Live Blog | 0 comments

Welcome to the Kernel Recipes Liveblog!

This is the 8th edition of kernel-recipes, and the second year with an official liveblog.

ftrace: Where modifying a running kernel all started — Steven Rostedt

ftrace is a tracer built into the kernel. It has multiple tracers, to trace functions, function graphs, etc. It supports many types of filters. Steven says that in addition to be the first framework to modify the runtime kernel, it was also the first to add glob filters inside the kernel.

To do modifications, ftrace relies on gcc's mcount profiler option (-pg). Instead of calling mcount, a trampoline calls into ftrace C code. For x86, it has since moved to "-pg -mentry" to remove the requirement for frame pointers.

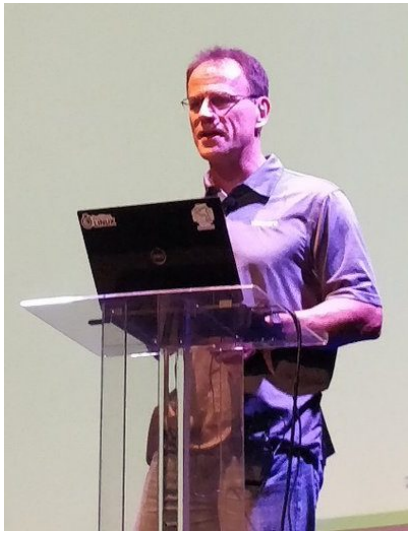
When doing the initial tests, with every function being profiled and calling an empty function, Steven found that the initial overhead was pretty big (13%). This lead to changing the build-time code to parse the ELF object files and finding each __fentry__. Then, with linker magic, adding magic variables to find each function start. At boot time, the kernel code will walk over the array of references, and replace every function entry trampoline with nops.

The ftrace overhead per function (struct dyn_ftrace) is about 64 bits, which takes 640kB of memory on Fedora 29.

One of this issue when modifying instructions at runtime is that it might work against the way the CPU fetches the code, caches it, etc. It took some time for Steve to find a proper solution: ftrace is now using breakpoints, and waiting for them to fire before modifying the entry instruction.

The ftrace_caller trampoline is used when there might be multiple functions that want a callback on given function entry, which are discovered at runtime; this can add some overhead. That's why the dynamic_trampoline was added for when there's only one caller, and it's hardcoded inside the trampoline. But the dynamic_trampoline caused complex issues with preemption; which is why the rcu_tasks was added by Paul E. McKenney in 3.18; which was then used by dyn_ftrace in 4.12.

Live patching also uses ftrace, with a small modification: instead of calling back into the original function, when the ftrace handler returns, it goes to the new patched function.



Analyzing changes to the binary interface exposed by the Kernel to its modules — Dodji Seketeli, Jessica Yu and Matthias Männich

The kernel ABI (kABI) is the low level binary interface between the kernel and its modules. It contains the set of exported symbols, their version, data structure layouts, etc.

In the upstream code, modversions/genksyms is the tools used to generate a view of the kernel ABI. Jessica says modversions is imperfect, and can easily trigger false positives of kABI breakages. That's because the output format is in text mode, and even identical definitions at the binary level might cause crc changes if the text definition varies a bit (for example, a comma instead of semi-colon in struct field separation). libabigail, the tool Jessica and her team are developing, does not have this type of issues since it works directly at the binary level.

Distributions usually want to do ABI tracking for just a subset of the exported symbols. They want human readable kABI reports, runtime ABI checks, and not have a too big impact on build time.

Libabigail, Dodji says, is a framework to analyze ABI by looking at binaries directly. It's a library that reads ELF and DWARF information, and builds an internal representation of ABI artifacts (functions, variables, types, ELF symbols). There a few specific tools built on top of libabigail.

The kernel has specific symbol tables, which needs specific support in libabigail: __ksymtab, __ksymtab_gpl. In addition, the kernel is very big, with thousands of modles, and hundreds of thousands of types (after deduplication).

There are multiple types of changes detected by libabigail: enum member sorting (changes the value of definitions); adding a struct member (changes the size of the structure, and order). Even small changes that don't break the kABI are

detected (but filtered out by default) : for example, if a struct member name changed.

If a type of a struct member changes, but not its memory layout, it's also detected.

In Android, there's an exploration to use a generic kernel image, with separation from the vendor bits, like it was done in userspace (Treble). Matthias is working on that, and using libabigail. This work is limited to LTS releases only; the goal is to have a single kernel configuration for all vendors, a single toolchain, and a limited subset of ABI scope.

libabigail is being integrated into the Android Kernel Build, with tooling to give generate an ABI report. This is also integrated into the Gerrit system to have the report at the time of patch submission to catch changes before they're even merged. libabigail ABI representation is output into XML files, which can be used by many tools.

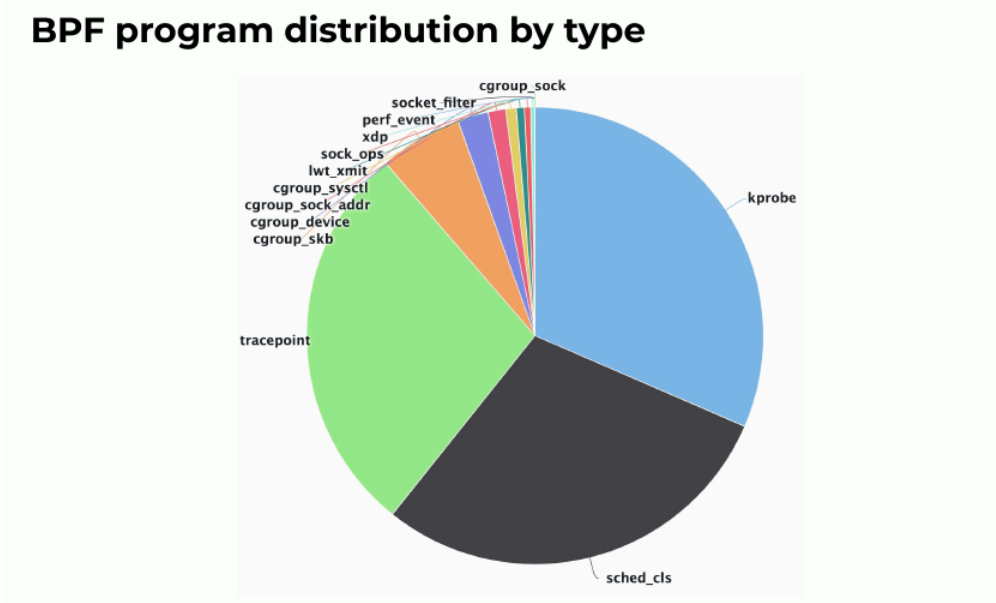
An example of unhandled case is an untagged enum: when enum values are used by a function, but the type (for example an `unsigned long flags` set) is not of the enum type. Another example, would be if such values are in `#define`s instead.

BPF at Facebook — Alexei Starovoitov

Alexei sent the first BPF patch a few years ago. He then joined facebook later to get more experience on using BPF at scale. Facebook has an upstream-first kernel policy for its datacenters, with close to zero patches being used (except those being tested before upstream submission).

Alexei reiterated the important Linux ABI rule: do not break userspace, even by modifying perceived behavior.

On a given laptop machine, running `ls -la /proc/*/fd | grep bpf-prog | wc -l` to count BPF programs would show already 6 being used at given time. It is widely deployed. At Facebook, they already have about 40 BPF programs being loaded at a given time on a server, with peaks at 100 for short period of time.



Alexei says that quite often, people point to BPF as a culprit when an issue happens. He gave a first example, with a daemon that did network capture and used BPF. It caused a 1% CPU regression, which was thought to be because of BPF. Doing tests without BPF and with nflog capture instead showed similar CPU usage, so it was easy to show BPF was not at fault.

In another example, a performance regression was caused when using a BPF tracepoint. But it was really at the kprobe setup time (before BPF program loading) that the regression happened.

The last problem was an example of another investigation which led to something completely unrelated. But the BPF tools were very useful to find the root cause (priority inheritance coupled with mm semaphone).

When using BPF in development, the BPF program will be built at runtime. This is the same on production servers; but BCC embeds LLVM, which has a big memory (.text) usage at rest, and even bigger at compile time. This led Alexei to work on a way to compile the program once and then reuse the artifacts on all server. That's why the BTF format was created to make sure the program did not need kernel headers to work. The BTF format embeds types, relocations, and some source code. It required work to add new clang/llvm builtins, which were upstreamed, too.

The BPF verifier improved a lot in 2019: bounded loops were added after two years of work. The `bpf_spin_lock` was added, which proves that no deadlock can happen. Alexei that the BPF verifier is even smarter than LLVM: it finds dead code even after llvm -O2, for example. Alexei says he is blown away this happens, even after a few years of being a compiler developer.

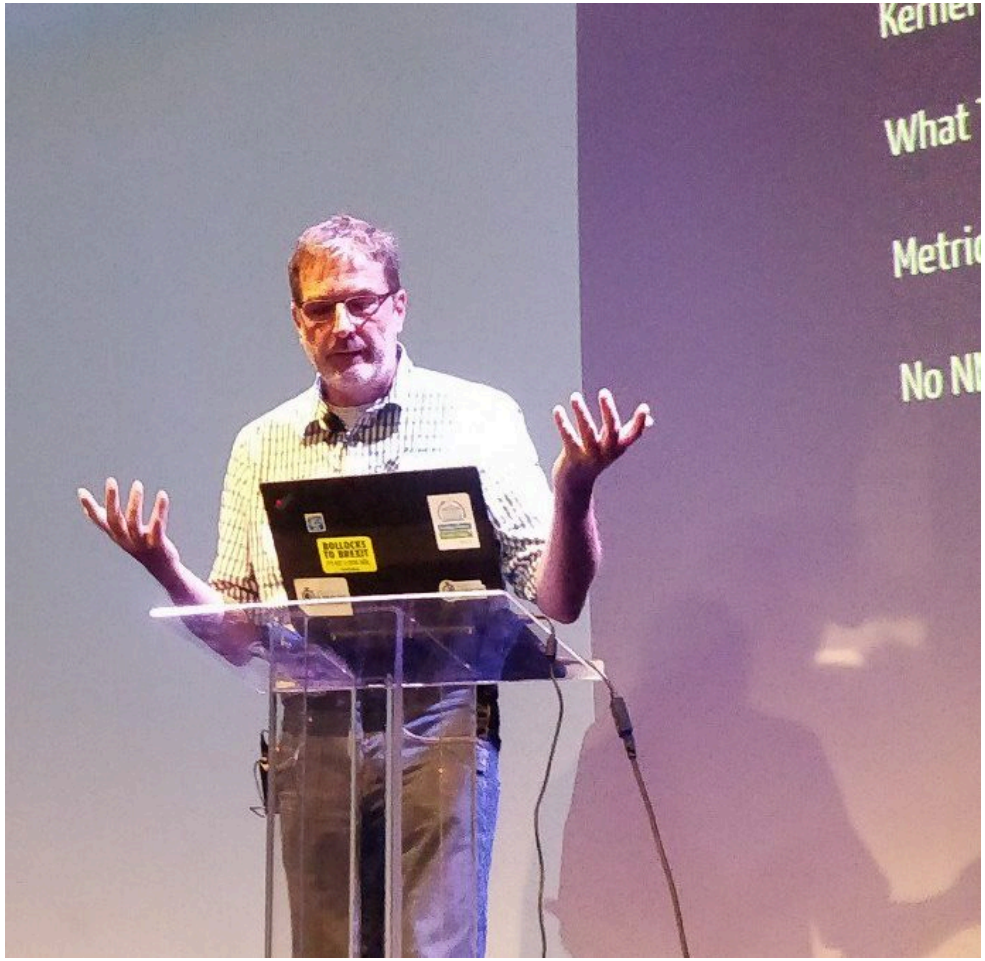
The verifier 2.0 can now be even smarter since it has in-kernel BTF information to use, and does not need to rely just on analyzing assembly code. BPF verifier does not trust userspace hints by default, and with the new BTF information, the dataflow analysis possiblities are on an entirely new level. A lot of hand-written and error-prone code in the verifier to validate particular BPF program types will be removable once the BTF option is enabled in the Linux kernel.

That's it for the morning talks! *Continue with the afternoon talks.*


BPF at Face

Live blog day 1 — afternoon

by Anisse ASTIER | Sep 25, 2019 | Live Blog | 0 comments



You can read the morning talks report here.

Kernel hacking behind closed doors — Thomas Gleixner

In the kernel, the security bugs are handled by the security team, which decides whom to bring to address specific issues, which would then be fixed and shipped within seven days. For a long time it worked relatively well, bugs were being fixed, and it was routine.

But two years ago, hardware security bugs appeared that had to be fixed in the kernel: it was the speculation attacks. Suddenly, the old way of doing didn't work: fixes had to be coordinated with other (OS) vendors. The playbook just didn't work anymore.

For Spectre and Meltdown, Intel took charge of the coordination between vendors; but multiple Incident Response teams inside Intel were handling different vendors, and they just could not communicate with each other. When public disclosure happened, all vendors had different patches in their trees with different type of brokenness.

When the Kaiser patches were posted, Thomas was tipped on the context of the speculation attacks, and forced to look at the patch series. After lots of cleaning, this turned into KPTI, which was merged partially very quickly, which felt to many as something very weird. And that was just for Meltdown, and Spectre was still around the corner.

When Spectre was disclosed, a few day before the deadline, three patches addressing it were posted. But they were all broken, Thomas says. Discussion was very confused on the mailing lists because of the lack of documentation. It was "panic engineering" as Thomas calls it.

That led him to send an email that to ask a few yes/no questions to understand all the implications, and then to reset all the panic by taking some time to go back into "normal mode".

Later, an agreement was reached between community of vendors for industry-wide collaboration, to have an upstream-first policy and no compartmentalization.

Unfortunately, LKML couldn't be a good forum for this type of discussion; after a small test of Keybase IO (an encrypted slack-like chat), the decision was taken to use encrypted mailing lists. After trying an existing project, he decided it was too buggy and un-patchable.

That's why Thomas started hacking on a small python project to handle encrypted emails — he had it up and running three days later handling S/MIME and PGP. It broke a few corporate mail servers by having the spam checker running out of memory. The mail servers would just crash, and the backlog would grow. After a few denial of services, the server was whitelisted.

His code evolved and was later published.

Unfortunately, the disclosure and handling was still controlled by Intel, and its lawyers only understood NDAs, which weren't workable in the kernel community. That's why a new formal process was created, with a few public Security officers (Thomas, Greg and Linus) that are Linux Foundation fellows.
It was designed exclusively for hardware security issues, with strict disclosure rules for kernel developers to avoid any potential conflict of interests across vendors; those rules were formalized in a Memorandum Of Understanding, that relied on trust.

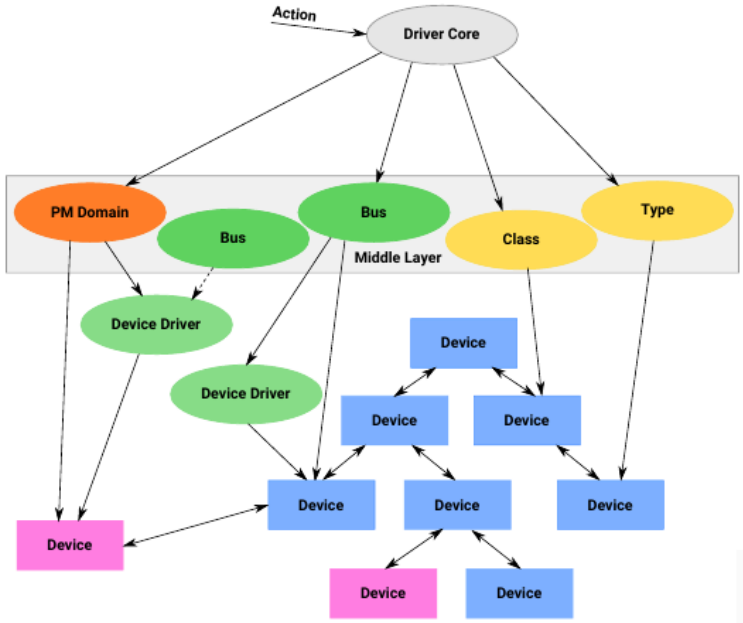
This process brought wide Industry acceptance, with all the major players agreeing. Thomas only hopes that it won't be necessary before he retires, but he knows that it's only wishful thinking.

What to do when your device depends on another one — Rafael Wysocki

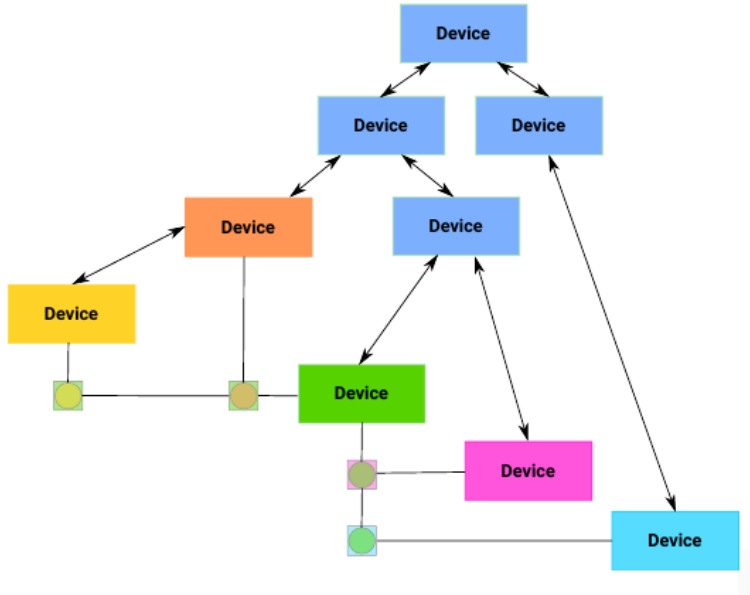
In a modern computer, or mobile phone, devices depend on one another.

The simple case is the Tree topology. In that case there's nothing to do, and the driver core will handle this automatically. Rafael showed a power management example: during suspend, all childen are suspended first, and during wakeup, it's the opposite.

But if there is a dependency that does not match the tree topology, it more complicated.



An example of a lightweight solution to this dependency problem is a device links data structure. This helps encode dependencies that aren't part of the tree:



This list should be in the proper order for shutdown ordering. This order should be separated from the driver probe order.

Another usecase for this is PM-runtime ordering. This is used at runtime to suspend unused devices in order to save power.

The third version of device links was merged recently in 4.10. But it still needed rework, and the latest updates should be in Linux 5.4.

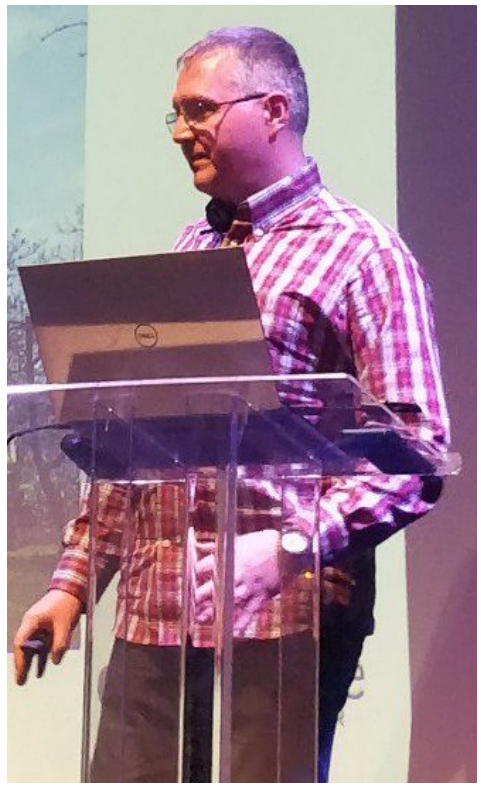
There are different types of device links; the managed device links allow the driver core to make decision on the probing and, removal or suspend of a given device, depending on the provided flags, and if a device is a consumer or supplier.

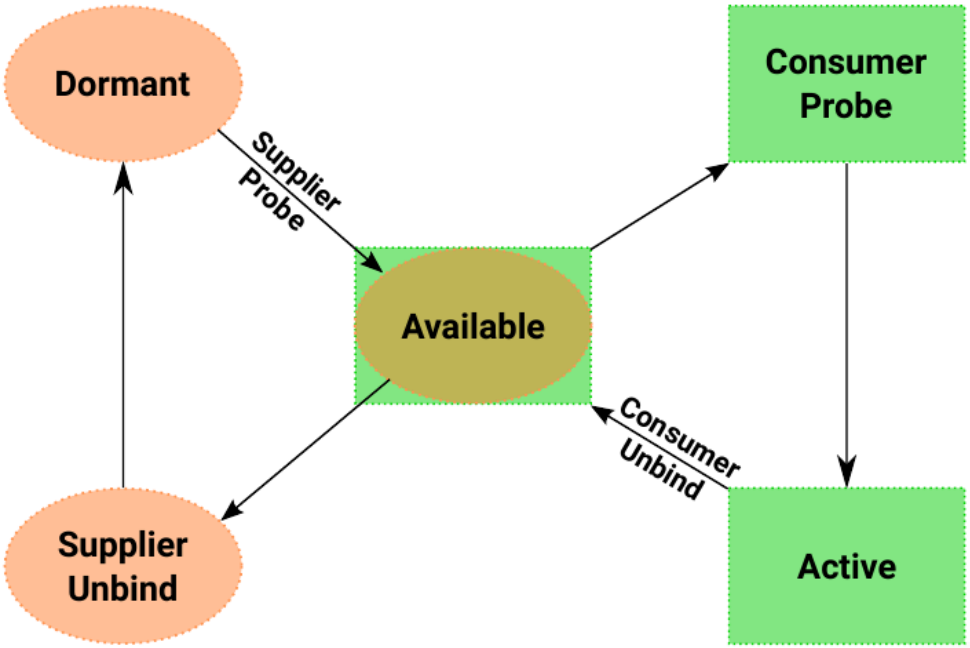
The stateless device links encode a type of dependency that doesn't cause the driver core to impact probing or suspend. It's a kind of soft dependency.

When adding a device link, only the supplier device needs to be registered. Circular dependencies are detected when adding a link (returns an error). Stateless device links require cleanup, while managed device links do not.

In order to add a dependency, one device driver can get a pointer to the supplier or consumer on the other side by using the Device Tree phandles, ACPI companion, or some other layer.

There is an internal state machine for the managed device links:





There are a few rules regarding device links flags that are documented. and should be followed depending on the type of device links. There are already users in-tree, and patchsets posted to build more features on top of device links, like complex driver dependencies.

Metrics are money — Aurélien Rougemont

Aurelien has been working in IT operations for 19 years, and has seen many bugs waking him during on-call. He wants to relate some of those.

The first one, starts with load average. It's a very complex thing to understand, especially since Linux has a very specific way of computing it.

A colleague of Aurélien added a new very common switch to the infrastructure, replacing an old home-made one. After adding monitoring to it, he noticed something very weird: more than 50% of packet loss. That's when Aurélien was called in for investigation. He found that it might be related to an old kernel bug that had been fixed, but they could not upgrade due to a binary driver. He dug into the documentation and finally found what was needed to do in order to fix the metrics collection. Unfortunately, based on wrong metrics, the network capacity had been upgraded in advance and cost the company 2.8M euros.

Another issue was a new SFP+ adapters which did not have proper ethtool metric. Aurélien wret a kernel patch to add those metrics, which helped find 480 faulty SFP+ in a new batch of hardware. They were sent back, saving the company 200k+ euros.

In a product Aurélien worked on, it used various disks with a ZFS raid array. But when receiving a new disk batch, Aurélien noticed a performance discrepancy between the disks: there were good ones, and bad ones. The bad ones were reported to the manufacturer, which after lots of fighting and sharing proofs of bad performance, the manufacturer agreed to change, saving the company 150k+ euros.

Aurélien's point is that metrics should be always used, but being careful of taking automated decisions based on those metrics: always look carefully if you're measuring what you want to be measuring.

In conclusion, Aurélien says that one shouldn't think that kernel code is sacred. Go read the code, the git history per subsystem, and don't be afraid to contribute, even if your first few tries will be bad. Even bad patches can help kernel developers understand what's missing from the documentation.

No NMI? No Problem! – Implementing Arm64 Pseudo-NMI — Julien Thierry

An IRQ is a manifestation of an event in the system: from a device or program. It interrupts the normal flow of execution.

A Non-Maskable Interrupt (NMI) is an interrupt that can still happens when interrupts are disabled. In x86, there's a dedicated exception entry for NMIs. In Linux, there's a specific context for NMIs with `nmi_enter()` and `nmi_exit()`; it informs system-wide features like printk, ftrace, rcu, etc. It puts a few restrictions in place: no NMI nesting, no preemption which help keep the NMI handlers simple.

In an NMI handler, it's not possible to use a spinlock, because they can't protect against interrupt: one should just use a mutex instead.

NMIs can be useful for perf. Julien showed an example on arm64, where during a long perf record, perf would use the performance record unit, and the IRQ restore code would be at the top of CPU usage. That's because, when restoring IRQs, all pending interrupts need to be processed, thus their impact being recorded at IRQ restore time.

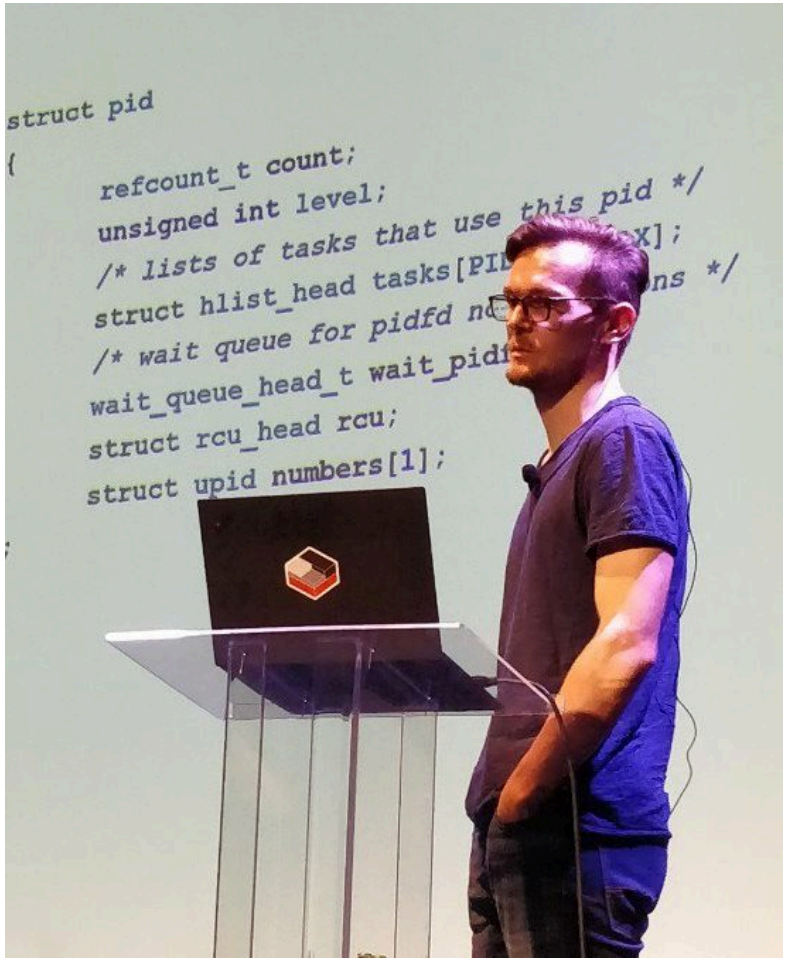
In arm64, interrupts are being handled by the Generic Interrupt Controller (GIC), which wakes up the CPU through the GIC PMR interface, which then sets the PSTATE.I bit. In order to implement NMIs in Linux, the goal is to stop touching PSTATE.I bit, and mask IRQs using PMR instead. The high priority NMIs would then be configured directly in the GIC.

Julien finally submitted an implementation of NMI interrupts upstream, which really helped perf profiling to be accurate of when events happened, during masked-interrupts contexts. The patches were merged in Linux 5.1, with important fixes in 5.3. It needs an aarch64 platform with GICv3 to work.

This liveblog is done for this first day. Read the second day posts here.

Second day — morning

by Anisse ASTIER | Sep 26, 2019 | Live Blog | 0 comments



pidfds: Process file descriptors on Linux — Christian Brauner

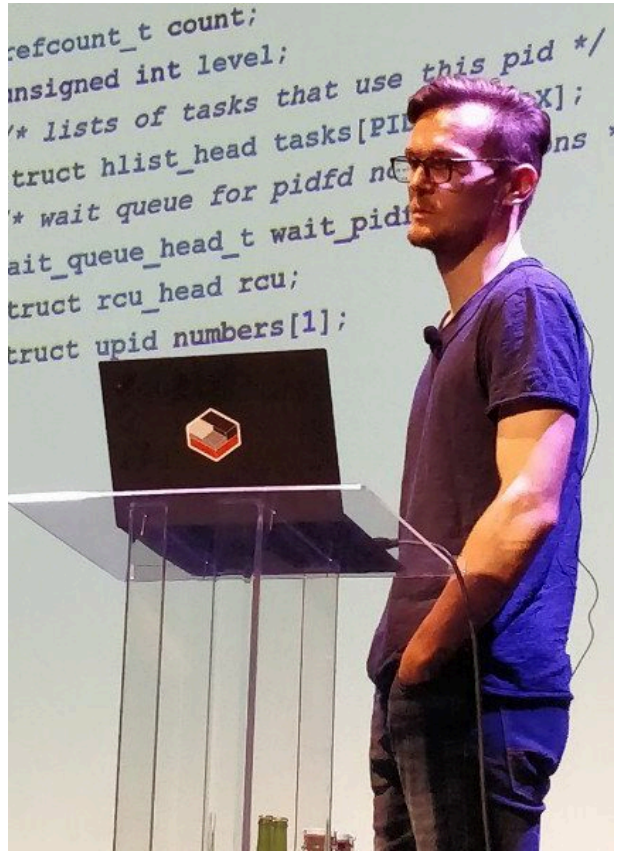
A pidfd is a file descriptor referring to a process. It is a stable, private handle that guarantees a reference to the same process (unilike PIDs).

Inside the kernel, pidfds use a pre-existing stable process handle: `struct pid`; it didn't reference `task_struct` because it's too big in memory.

The main goal was to avoid the pitfalls of pid recycling on high-pressure systems, which caused a few security issues over the years: this is when the process ID numbers are being reused, because the pool (usually 32k) has been fully used, and this causes race conditions.

Anoter reason is linked to the way shared libraries want to handle subprocesses without interacting with user processes's process management (SIGCHLD). It's also helpful for handing process management to a non-parent with fd-passing. FDs are well known and many userspace programs know how to handle them.

There are already many userspace programs that care about pidfds: dbus, qt, systemd, criu, etc. This isn't the first time an OS implements file descriptors for process IDs: Illumos has an userspace emulation of this feature, FreeBSD



has `pdfork()`, `pdgetpid()` and `pdkill()`. There were also a few proposals before for Linux that weren't merged: forkfd and CLONE_FD, both of which Christian looked at to understand why they didn't land.

In Linux 5.1, signal support with pidfds was added for reliably sending signals to processes. Lots of people had opinion on this, in particular, many people wanted to be able to use pidfds with files from `/proc`. This is now possible, but not completely race-free, Christian says. It's not an ideal solution, but it worked.

In Linux 5.2, the `CLONE_PIDFD` flag was added to obtain race-free pidfds at process creation time. The way it was implemented was initially controversial because Linus initially wanted fds from /proc, but Christian was pushing with anonymous inodes. After implementing both of them, it was obvious that the latter would be simpler and easier to maintain. A nice bonus of pidfds is that they are `O_CLOEXEC` by default, meaning they will be closed automatically when the process calls an `exec()`-related syscall.

In Linux 5.3, the `clone3` syscall was added, and it has a dedicated argument for pidfd instead of abusing other arguments like clone2. Polling support was also merged to get process exit notification for non-parents that have a pidfd.

In 5.3, `pidfd_open()` was also added to create a pidfd after creation time (without `CLONE_PIDFD`). In 5.4, a new `P_PIDFD` flag was added to `waitid()` to wait for a process through a pidfd.

Current work in progress include kill-on-close semantics, to send a SIGKILL to a process when the last fd referencing it is closed. Another semantics that's being thought of is a way to have exclusive waiting to hide process from generic wait requests: it would be a flag at clone time. Christian is also thinking about a way to use pidfds for some namespace management tasks, but the scope and security impact isn't clear enough yet.

To conclude, Christian says that resilience is important: understanding what reviewers want, what is important, and what is bikeshedding is a critical skill.

Keeping the kernel relevant with BPF — David Miller

The kernel has changed over the years, David says. It used to be that known breaking changes could be merged quite quickly, but this doesn't happen anymore.

Nowadays, one should always explain the use cases, think hard about the API impact and its extensibility. When writing a kernel change, it always takes time to write the test, take reviews into account, and iterate.

David argues that in order to propose syscalls and design them properly, you must be arrogant. You are putting the users inside the boundaries of your design, and making choices for them. This isn't necessarily what users want: they always want maximum flexibility, fast iteration, or to have an arbitrary policy for example.

Describing BPF is complex, because it pushes this maximum complexity to users. BPF provides a mechanism by which users can solve their problem more freely. BPF also seems contradictory because it gives users maximum freedom, but it also provides containment and safety.

That's why, David says, people understanding BPF should go and speak about it so that everyone understands its impact on kernel development and user freedom.

Hunting and fixing bugs all over the Linux kernel — Gustavo A. R. Silva

Gustavo started contributing to the kernel in 2013, which is quite recent, he says. He has since been fixing bugs all over the kernel.

Part of his work starts with analyzing the reports from the proprietary static analyzer Coverity. It gives a lot of false positives with the kernel, which takes Gustavo a lot of time to review. He still committed more than 500 fixes thanks to Coverity over the years. A few high impact issues were found with it, like unitialized memory use or out-of-bounds access.

He says he looks at every issue; the scan used to run on weekly tagged -rc releases, but now he has access to daily scans, including from linux-next, so that bugs are caught before reaching mainline.

He has many examples of bug fixed. The first one is a wrong variable type, where a counter's max value was raised from 100 to 1000, but the counter was an uint8_t, so its maximum type value was 255. The fix simply changed the type to uint16_t.

Many fixes look trivial in retrospect, but they catch real bugs. Another one was inconsistent use between IS_ERR and PTR_ERR variables. It was caught with Coverity, but Coccinelle can also catch this type of issue.

Gustavo has found issues in linux-next, before they hit mainline, like a wrong use of bitwise operators. When investing missing fallthrough, he's found missing returns. He has found resource leaks (missing goto). He found a seven years old bug in a perf test, or an 8 year old bug in an USB gadget driver.

Gustavo has collaborated with the kernel self-protection project to help remove Variable Length Arrays. He also helped introduce the struct_size helper, that helps compute the size of structure that has a tail array with a variable size at allocation time.

This struct_size helper macro helped not only simplify the code, but also catch buffer overflows or undefined behaviors in some places.

A big work Gustavo did this year, was to fix the -Wimplicit-fallthrough warnings. This helps find instances where the a break has been forgotten in a switch. If the fallthrough is intentional, a comment should mark it to fix the warning. Gustavo reviewed thousands of cases (2300 initially just for x86) throughout the code, and sent numerous patches; sometimes for quite old bugs. He initially encountered a lot of resistance, but once he started having successes, more maintainers understood the need to fix the warnings, and his patches went in more easily.

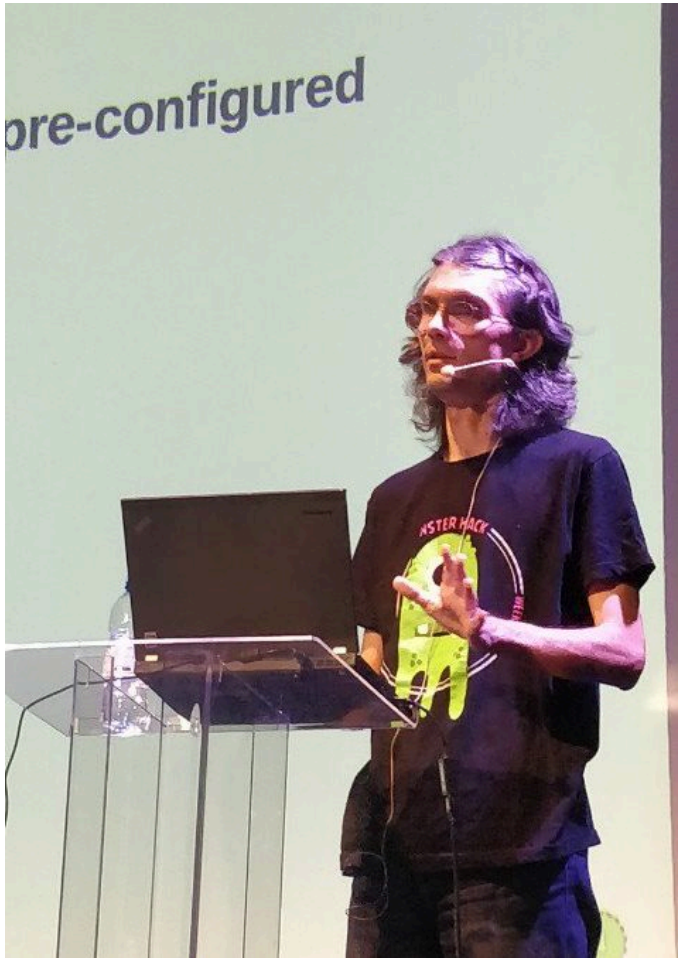
He spent part of his time working on this for long time, but he says it was all worth it the end: the warning has been enabled by default in Linux 5.3, and people already started catching bugs with it.

Gustavo says he needed to have his own tree to fix the last 10% of issues were patch being ignored, sometimes deliberately.

That's it for this morning ! Continue with the afternoon report.

Second day — afternoon

by Anisse ASTIER | Sep 26, 2019 | Live Blog | 0 comments



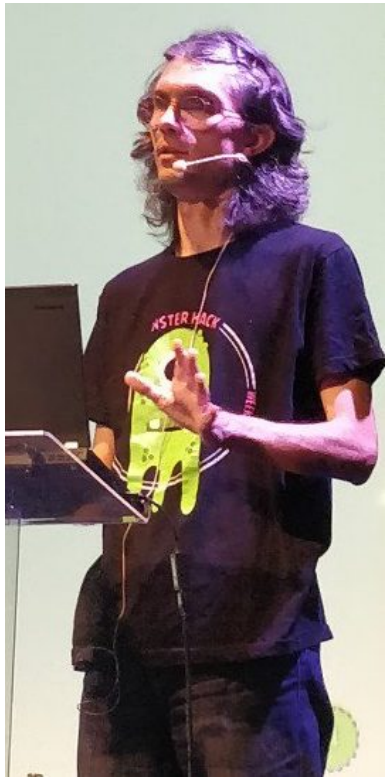
Marvels of memory auto-configuration (SPD) — Jean Delvare

Serial Presence Detect is a standard used to configure memory.

In the beginning, memory was soldered on the boards directly. The first DRAM module was a 30-pin SIMM from 286 to 486. Everything was hardcoded since there was place for configuration information on it. Then came PPD (Parallel Presence Detect), where 4 pins were used for configuration. It quickly became not enough. and hardware probing or manual configuration was still needed.

Later PPD revisions added more pins, but it didn't solve the problem nor scale. The need to finer-grained timings information also grew with time, with manual configuration getting more and more complex, to be close to impossible.

When the first DIMM module arrived with the first Pentium generation, SPD replaced PPD. Only 5 pins were needed to configured the data, with serial buses on it, which helped reduce the board footprint. The data was stored in cheap EEPROMs, and all the information was available in up to 256 bytes.



The EEPROMs were compatible with SMBUS and I²C, which means you can use i2cdetect to find them on DDR3 motherboard.

With DDR4, there was a need to have more space to store config parameters. Using bigger EEPROMs have been evaluated, but it proved to bring compatibility issues. So Jedec (the DRAM standard body) decided to define a new EEPROM standard for DDR4. This new standard is named EE1004, and is pin-compatible with the previous standard.

This new standard supports write protection (per block), but is much more complex, Jean found. It abuses the i2c protocol to send commands in addresses, it uses broadcast messages for everything, and isn't extendable. The spec also includes useless payloads in messages. Since the vendor spec is "crap", memory modules implemented "crap" on top of it, as Jean says.

Linux support is not essential, Jean says, since in the normal case, only the BIOS or firmware needs to initialize memory. But it's useful for diagnostics, inventory or knowing the exact memory module to add a similar one. Jean says that dmidecode (which he also maintains) is just not accurate enough on the memory modules.

Drivers for the SMBUS controller and the EEPROMs themselves are needed first for Linux support. There are then userspace tools, the main one which Jean also maintains: decode-dimms. It's a perl scripts that parses the information of all DDR to DDR4 EEPROM information.

An issue we have today in Linux is that the EEPROM devices need to be instantiated before the SPD information can be fetched and parsed. Jean is working on adding automatic probing based on DMI data and I²C probing on the SMBUS, in order to automatically instantiate the appropriate device on responsive addresses.

XDP closer integration with network stack — Jesper Dangaard Brouer

XDP was needed for the kernel networking stack to stay relevant, amidst the emergence of kernel-bypass technologies. Jesper's goal is to stay within 10% of similar kernel bypass technologies.

It's designed as a new, programmable layer in front of the network stack, allowing to read, modify, drop, redirect or pass packages before they go into the kernel network stack.

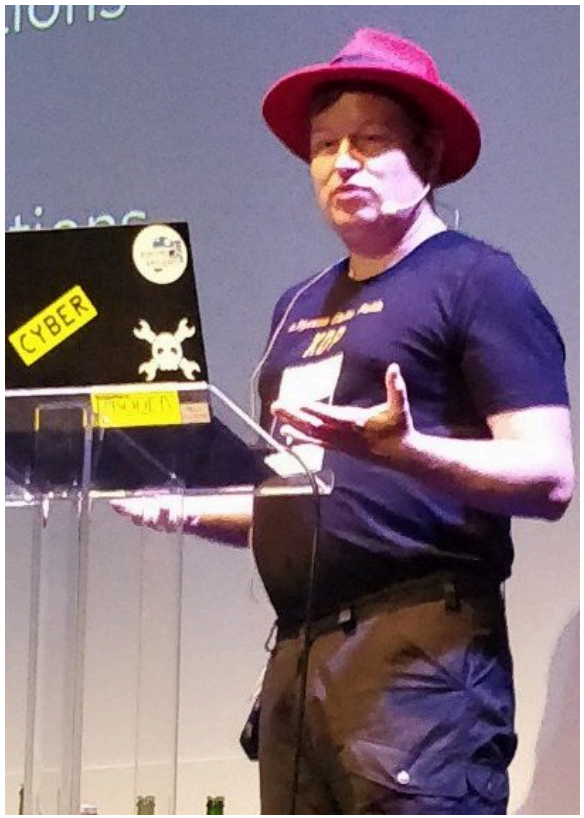
AF_XDP is used to provide kernel bypass of the netstack. DPDK, a competing technology already integrates a poll-mode for AF_XDP. The advantages of AF_XDP, is that it allows sharing NIC (network card) resources.

XDP is faster because it bypasses allocation, and lets you skip kernel network code. But skipping the network stack means you can't benefit from its features. So it might be needed to re-implement them. To avoid this reimplementation, XDP enables adding BPF-helpers instead to share resources.

Jesper want people to see XDP as a software offload for the kernel network stack. An example of work in this area is accelerated routing or bridging in XDP, but with fallback to the kernel network stack for non-accelerated packets.

One goal of Jesper is to move SKB allocations outside of NIC drivers, in order to simplify the drivers, by creating the SKB inside the network-core code, this already works with xdp_frame and the veth driver and cpumap BPF type. But there are still a few roadblocks for that with various types of hardware offloads used by SKBs that need to be taken care of.

Jesper has a few ideas on how to solve this, and he wants to add a generic offload-info layer, and is looking for input on how to implement it. The generic offload-info also needs to be dynamic so that each driver has its own format for hardware



description.

Another issue for network-core SKB alloc is to have XDP handling for multi-frame packets. Jesper says that this issue has been ignored for a while, but that it's time to have proper handling for the multi-frame packets use cases: jumbo frames, TSO, header split.

Having generic xdp_frame handling has a few advantages: it can help add a fastpath for packet forwarding for example.

Many people complain that XDP and eBPF are hard to use. That's why Jesper created a hands-on tutorial with a full testlab environment, that he releases today at Kernel Recipes.

Jesper says that XDP is missing a transmit hook. This is more complex than it sounds, because it needs a push-back and flow-control mechanic, but without reimplementing the qdisc layer that already does that in Linux.

In conclusion, there are a few areas that still need work in XDP, and Jesper says that contributions are welcome to address the many needs.

Faster IO through io_uring — Jens Axboe

The synchronous I/O operations syscalls in Linux are well known, and evolved organically. Asynchronous operations with aio and libaio don't support buffered I/O, O_DIRECT isn't always asynchronous and aio is generally syscall-heavy, which is quite costly in a post-speculation mitigations era.

This lead to Jens to write a wishlist of asynchronous I/O features. He wondered if it was possible to fix aio. After a fair amount of work, he gave up and designed his own interface: io_uring.

He met quite a bit of resistance from Linus, who was burned with previous asynchronous and direct I/O attempts that have little usage. It was still merged in 5.1, with more features added in recent releases.

Fundamentally, io_uring is a ring-based communication channel. It has a submission queue (sqe), and a completion queue (cqe). It's setup with the `io_uring_setup` syscall that returns a ring file descriptor, and takes a big struct with parameters.

Ring access is done through memory mapping. Reading and writing the ring works with a head and tail indexes that are common to ring data structures.

To send a new sqe, one reads the current tail, verifies if the ring won't be full, writes data just after the tail, then updates its index in two steps with write barriers between each. For cqe, it works a bit similarly.

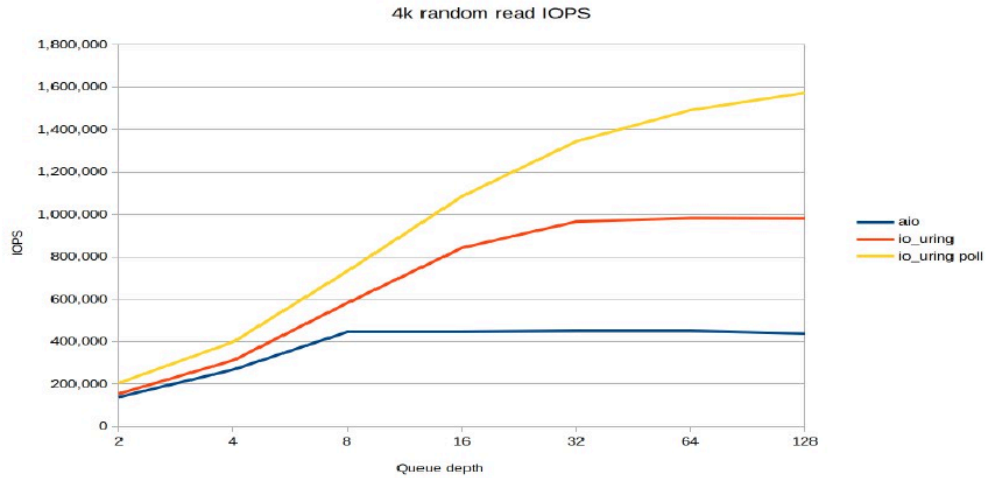
To submit requests, one uses the `io_uring_enter` syscall, which enables both submit and complete at the same time.

Jens says that if this looks a bit complex, it's because it's not made to be used directly in an app. There's `liburing` for that which handles the complexity of ring access behind the scenes.

liburing has various helpers to the multiple commands supported by io_uring (read/write/recv/send/poll/sync, etc.). It has a few features: it's possible to wait for the previous command to drain. It's possible to link (chain) multiple commands, like write->write->write->fsync. There's also an example to do direct copy of an arbitrary offset within a file to another within another file in examples/link-cp.c.

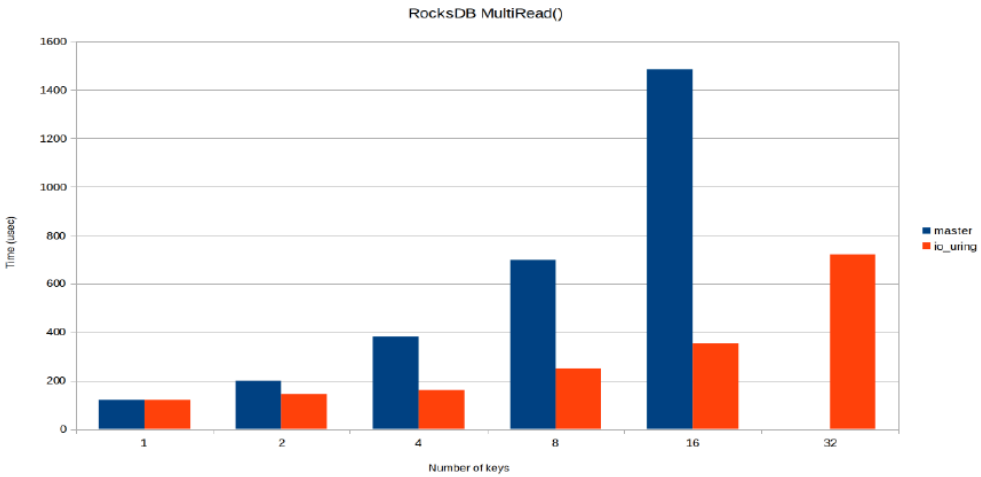
It supports multiple features based on registering aux functions for advanced features, like eventfd completion notifications.

Polled IO is useful when wants to trade CPU usage for latency. It's supported automatically by ioring at setup time, but can't be mixed with non-polled IO. It's also possible to do polled IO submission where reaping is app-polled in order to do I/O without a single syscall.



In addition to having better raw IOPS and buffered read performance, io_uring also has better latency.

Adoption is really growing, with interest in the Rust and C++ communities, in Ceph, libuv and postgresql. In RocksDB, the MultiRead() performance greatly improved:



A simple single thread echo server written by @CondyChen showed better performance than with epoll. Intel has also had great NVMe improvements on IOPS from 500k to 1-2M IOPS when using io_uring vs aio.

In the future, Jens imagines any high perfomance syscall that needs to be fully async could be implemented on top of io_uring. Jens maintains a definitive documentation of io_uring, which is being updated regularly.

The next steps toward Software Freedom for Linux — Bradley Kuhn

Bradely first downloaded Linux in 1992, and didn't boot it before 1993. Although he because a Free Software activist, he started as a computer science student who built a Linux-based lab in 1993. He had to patch the kernel to remove ctrl-alt-del to prevent any local user from rebooting the machine and disconnecting remote students.

That's when he did this small modification by using the fundamental freedom provided by the GPL which Linux used, that he understood how important it was.

The freedom the first kernel developers had to hack on their own devices led to a very useful OS. The first Linksys WRT router used Linux because of this. But it was also the first massively sold GPL-violating device. Once this violation was resolved, this lead to the creation of the OpenWRT community, which gave even more freedom to the users.

With time, more routers were released violating the GPL, which Harald Welte helped resolve, sometimes with litigation. This helped the OpenWRT project being ported to even more projects.

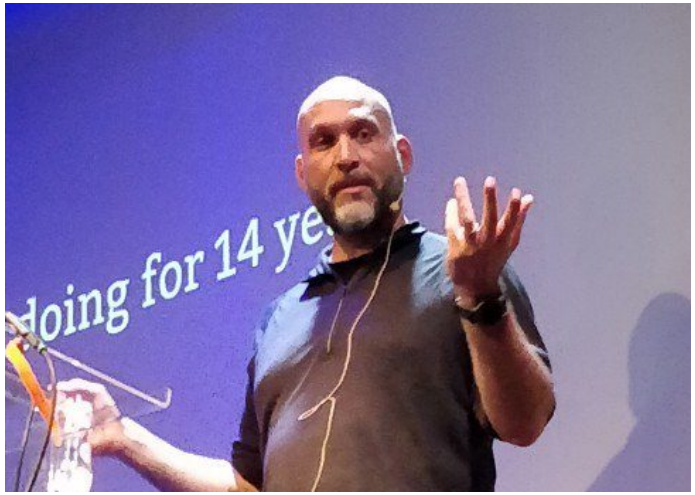
Bradley says that Linux is the most successful GPL software project out there. But to stay there, it needs users to be able to continue installing their modifications and tinkering.

GPL enforcement is a necessary part for that, Bradley says. Although he doesn't support malicious enforcers that want to get rich out of it. He says that Software Freedom Conservancy is available to help do ethical GPL enforcement.

That's it for today ! Conitinue with the last day live coverage.

Last day — morning

by Anisse ASTIER | Sep 27, 2019 | Live Blog | 0 comments



This is the last day of our liveblog coverage of Kernel Recipes. Enjoy !

Suricata and XDP — Eric Leblond

Eric has been working with networking for quite a long time, but it didn't help with his imposter syndrome, he says.

Suricata is an IDS/IPS: it analyzes network traffic from a probe and raises alerts. It has an advanced application layer analysis: it can do TCP payload re-assembly for example.

Suricata works as passive packet sniffer. It needs to work at the speed of the network, because it has no influence on network traffic. It must not miss packet, even under load. Eric says that at 3% packet loss, 10% of the alerts are missed; at 5.5% of loss, 50% of the alerts are missed.

It therefore needs a lot of CPU power, as well a proper way to load balance input network flows to CPUs: a scheduling issue can cause packet losses. There are multiple bypass strategies to mitigate that: local bypass works in userspace, but does not bypass the kernel. Capture bypass works before the netstack in the kernel.

Suricata also allows dropping traffic that is too intensive (e.g Netflix), or that it can't understand (post-handshake TLS). The bypass implementation started with NFQ, then moved to AF_PACKET XDP and eBPF.

The eBPF bypass implementation relies on libbpf. Flow detection was based on an eBPF flow table, that would be managed periodically. Unfortunately, it was very slow because it was syscall (and CPU) intensive.

He then worked with a Netronome card, which is a card that can run XDP eBPF code directly. It handles the eBPF maps directly; but there were a few constraints related to the hardware that were dealt with. He even used Netronome RSS load balancing to distribute packets on the proper queue to share load; all done in eBPF code on the card.

Suricata startup is also challenging, when analysis starts in the middle of most network flows. To help with restart, eBPF maps would be pinned in order to keep the working in-memory data between Suricata sessions.

XDP is used in Suricata to do tunnel decapsulation (e.g GRE) by splitting the inner data, so that it can be load-balanced properly. Suricata does TLS handshake analysis, but wants to bypass the encrypted data. Initially this was done at the flow level, but it caused issues because there many small sessions. It was then done with XDP pattern matching.

AF_XDP was very interesting for the Suricata use case, because it provided faster analysis than AF_PACKET, bypassing the netstack's skb creation for example. But Eric discovered that the hardware timestamp was missing with AF_XDP, which isn't workable with Suricata that needs it for packet re-ordering between NICs.

Eric says that the situation has greatly improved since it started in 2015, with libbpf now being the standard tool to interact with eBPF, and it is shipping in most distributions. He wishes that there would be common libraries to share eBPF protocol decoding. Situation could also improve with shipping of eBPF files, that need to be compiled on the production system.

CVEs are dead, long live the CVE! — Greg Kroah-Hartman

Greg says that the 40 minutes rant in this talk is really his personal opinion only.

A CVE is a unique identifier for a security issue. It's a dictionary, not a database. It started because there wasn't any way to identify a particular issue; cgi plugins and zlib had a lot of issues that pushed for a way to identify them.

The CVE format is CVE-YEAR-NUMBER, like CVE-2019-12379, that is unique per issue. Multiple vendors can hand them out, but not the Linux kernel security team.

Then there is the NVD, a database of CVEs, that gives them scores. The issue is that it's slow to update. The Chinese NVD (CNNVD) is faster to pick up CVEs, but doesn't update them.

The issue with the NVD is that misses some issues. It is run and funded by the USA government, which might cause issues of conflict of interest, Greg says.

Greg says there is no mapping between CVEs and patches. Spectre has 1 CVE entry, but hundred of patches. Or sometimes, a single kernel patch have multiple CVEs. The NVD does not point to fixes.

An issue with CVEs that they are abused by developers to pad their resumes. Greg showed the example of CVE-2019-12379 that affected his code, that was just simply wrong: there was a patch to fix an issue in case of kmalloc failing. But it was later found by Ben Hutchings that the patch was worse than the initial behaviour. Also, despite the patch being reverted an never reaching a public kernel.org release, the CVE had to be disputed by Ben, with a Medium score or a non-exploitable issue.

Greg is also saying that sometimes it's abused by Enterprise distribution engineers that only backport security fixes, to backport bug fixes instead that otherwise would pass through management.

Greg says that every week, there are lots of fixes that have security impact that aren't attributed a CVE; anyone running a distribution kernel that only backports CVE fixes, is insecure to known issues.

He gave an example of a "popular phone", running a Linux 4.14.85 kernel with a few backports. But since they are missing 1700+ patches, they missed 12 documented CVEs, as well as dozens of other security issues.

At Google, they found out that 92% of kernel security issues that were reported were already fixed in LTS kernels. That lead a change in policy to require updating the kernels from the LTS branch. They are still issues with out-of-tree code.

How to fix CVEs ? Greg says the best course of action is to ignore them. And build something new. It does need to have a distributed unique identifier. Greg says that upstream git sha1 commits ids already worked and are tracked.

Maybe they need better marketing ? Greg proposes to call those Change IDs: CID. The format would be CID-[12 digit hex numbers]. For example CID-7caac62ed598. Rebranding with CID identification is the way forward for open projects according to Greg.

Driving the industry toward upstream first — Enric Balletbo i Serra

Enric is a contributor to Chrome OS ARM platforms as part of his work at Collabora.

His dream is that one time a SoC vendors would not give him a BSP, and instead point to upstream Linux or u-boot as the code to use.

It's easier to work behind closed doors, Enric says. Working upstream takes time and patience. But it also brings higher quality and less maintenance.

Chrome OS is the Linux-based OS inside every chromebooks. Security is important for Chrome OS, Enric says: they are supported for 6 years, and updated every 6 weeks. Within these constraints, it's impossible to have a different kernel for every device. Right now, it works by using 5 to 6 LTS branches per hardware family, and attempting to update to a new LTS at least once during the lifetime of a device.

Chrome OS development happens in the open, unlike Android, Enric says. To help with development, kernel commits are tagged to identify upstream, backported, or chromium-specific code. A lot of commits are backported to the chromium kernel branches.

A requirements for hardware vendors with Chrome OS is to work with upstream. Enric showed an example with Mediatek and Rockchip that have started sending patches upstream for chromebook support.

In chromebooks, there is a CrOS-specific Embedded Controller (EC); its upstream support was incomplete, but Enric has been working on improving it. He worked by getting the patches from CrOS branches, squashing and splitting them before submission.

Sometimes, it would be accepted, but others it had to be reworked: for example, the use of the MFD (multi-function devices) subsystem was being abused. Now, most of the CrOS EC drivers are upstream. New related drivers will go through upstream first.

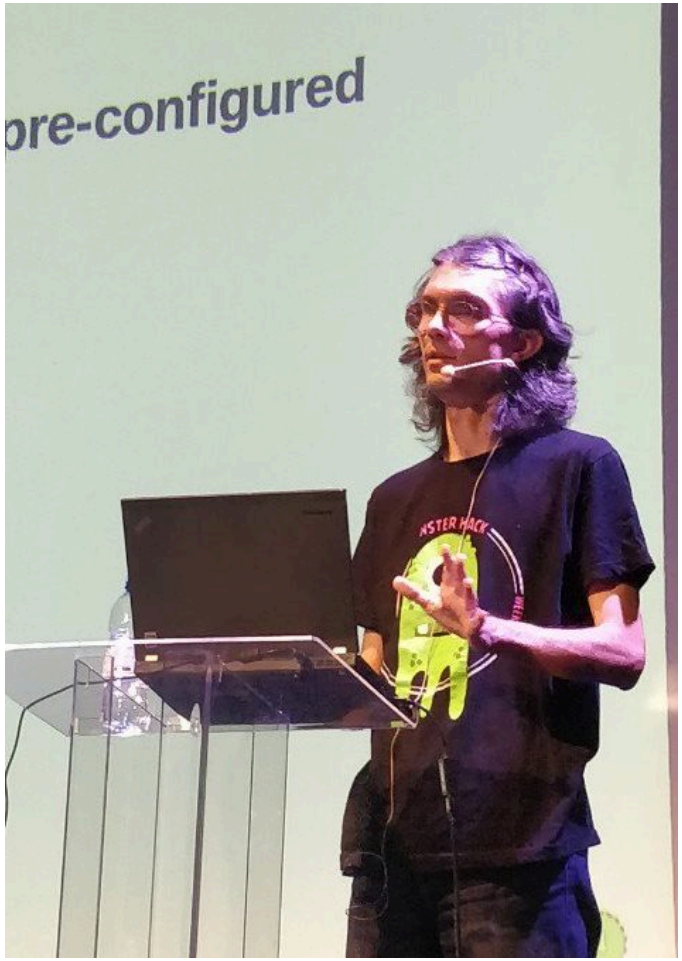
What's different from mainline in CrOS kernel code ? Mostly gpu and network drivers. The rest is part of a long tail. For net, it's because of a lot of the code was backported. For gpus, the midgard driver patches is the biggest source of patches. It's because when it was released, no alternative existed. But now, the panfrost driver for the same mali GPUs is upstream.

Enric showed a demo with a Samsung Chromebook Plus with a Rockchip SoC and an upstream Debian distribution. A lot of hardware is already supported, and the GPU as well.

That's it for this morning ! Continue with the afternoon liveblog.

Second day — afternoon

by Anisse ASTIER | Sep 26, 2019 | Live Blog | 0 comments



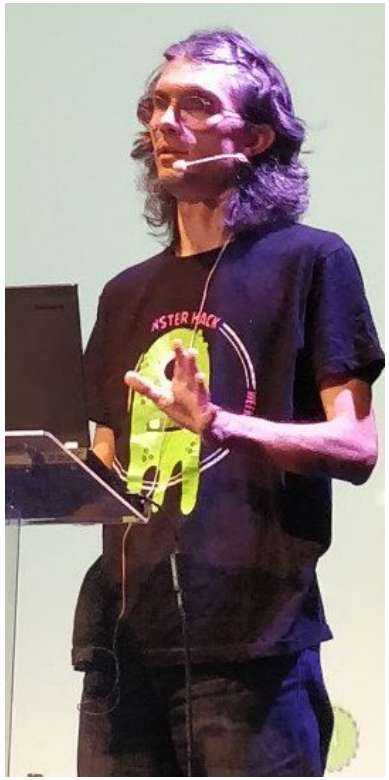
Marvels of memory auto-configuration (SPD) — Jean Delvare

Serial Presence Detect is a standard used to configure memory.

In the beginning, memory was soldered on the boards directly. The first DRAM module was a 30-pin SIMM from 286 to 486. Everything was hardcoded since there was place for configuration information on it. Then came PPD (Parallel Presence Detect), where 4 pins were used for configuration. It quickly became not enough. and hardware probing or manual configuration was still needed.

Later PPD revisions added more pins, but it didn't solve the problem nor scale. The need to finer-grained timings information also grew with time, with manual configuration getting more and more complex, to be close to impossible.

When the first DIMM module arrived with the first Pentium generation, SPD replaced PPD. Only 5 pins were needed to configured the data, with serial buses on it, which helped reduce the board footprint. The data was stored in cheap EEPROMs, and all the information was available in up to 256 bytes.



The EEPROMs were compatible with SMBUS and I²C, which means you can use i2cdetect to find them on DDR3 motherboard.

With DDR4, there was a need to have more space to store config parameters. Using bigger EEPROMs have been evaluated, but it proved to bring compatibility issues. So Jedec (the DRAM standard body) decided to define a new EEPROM standard for DDR4. This new standard is named EE1004, and is pin-compatible with the previous standard.

This new standard supports write protection (per block), but is much more complex, Jean found. It abuses the i2c protocol to send commands in addresses, it uses broadcast messages for everything, and isn't extendable. The spec also includes useless payloads in messages. Since the vendor spec is "crap", memory modules implemented "crap" on top of it, as Jean says.

Linux support is not essential, Jean says, since in the normal case, only the BIOS or firmware needs to initialize memory. But it's useful for diagnostics, inventory or knowing the exact memory module to add a similar one. Jean says that dmidecode (which he also maintains) is just not accurate enough on the memory modules.

Drivers for the SMBUS controller and the EEPROMs themselves are needed first for Linux support. There are then userspace tools, the main one which Jean also maintains: decode-dimms. It's a perl scripts that parses the information of all DDR to DDR4 EEPROM information.

An issue we have today in Linux is that the EEPROM devices need to be instantiated before the SPD information can be fetched and parsed. Jean is working on adding automatic probing based on DMI data and I²C probing on the SMBUS, in order to automatically instantiate the appropriate device on responsive addresses.

XDP closer integration with network stack — Jesper Dangaard Brouer

XDP was needed for the kernel networking stack to stay relevant, amidst the emergence of kernel-bypass technologies. Jesper's goal is to stay within 10% of similar kernel bypass technologies.

It's designed as a new, programmable layer in front of the network stack, allowing to read, modify, drop, redirect or pass packages before they go into the kernel network stack.

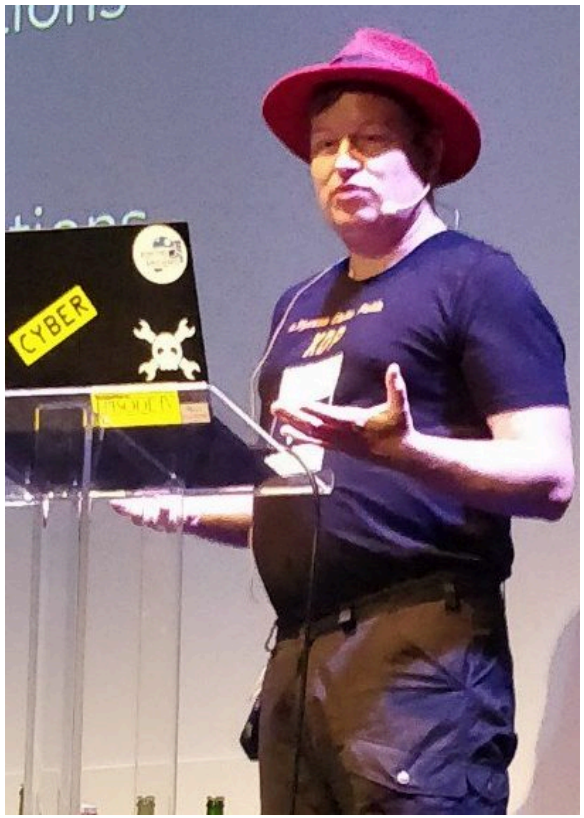
AF_XDP is used to provide kernel bypass of the netstack. DPDK, a competing technology already integrates a poll-mode for AF_XDP. The advantages of AF_XDP, is that it allows sharing NIC (network card) resources.

XDP is faster because it bypasses allocation, and lets you skip kernel network code. But skipping the network stack means you can't benefit from its features. So it might be needed to re-implement them. To avoid this reimplementation, XDP enables adding BPF-helpers instead to share resources.

Jesper want people to see XDP as a software offload for the kernel network stack. An example of work in this area is accelerated routing or bridging in XDP, but with fallback to the kernel network stack for non-accelerated packets.

One goal of Jesper is to move SKB allocations outside of NIC drivers, in order to simplify the drivers, by creating the SKB inside the network-core code, this already works with xdp_frame and the veth driver and cpumap BPF type. But there are still a few roadblocks for that with various types of hardware offloads used by SKBs that need to be taken care of.

Jesper has a few ideas on how to solve this, and he wants to add a generic offload-info layer, and is looking for input on how to implement it. The generic offload-info also needs to be dynamic so that each driver has its own format for hardware



description.

Another issue for network-core SKB alloc is to have XDP handling for multi-frame packets. Jesper says that this issue has been ignored for a while, but that it's time to have proper handling for the multi-frame packets use cases: jumbo frames, TSO, header split.

Having generic xdp_frame handling has a few advantages: it can help add a fastpath for packet forwarding for example.

Many people complain that XDP and eBPF are hard to use. That's why Jesper created a hands-on tutorial with a full testlab environment, that he releases today at Kernel Recipes.

Jesper says that XDP is missing a transmit hook. This is more complex than it sounds, because it needs a push-back and flow-control mechanic, but without reimplementing the qdisc layer that already does that in Linux.

In conclusion, there are a few areas that still need work in XDP, and Jesper says that contributions are welcome to address the many needs.

Faster IO through io_uring — Jens Axboe

The synchronous I/O operations syscalls in Linux are well known, and evolved organically. Asynchronous operations with aio and libaio don't support buffered I/O, O_DIRECT isn't always asynchronous and aio is generally syscall-heavy, which is quite costly in a post-speculation mitigations era.

This lead to Jens to write a wishlist of asynchronous I/O features. He wondered if it was possible to fix aio. After a fair amount of work, he gave up and designed his own interface: io_uring.

He met quite a bit of resistance from Linus, who was burned with previous asynchronous and direct I/O attempts that have little usage. It was still merged in 5.1, with more features added in recent releases.

Fundamentally, io_uring is a ring-based communication channel. It has a submission queue (sqe), and a completion queue (cqe). It's setup with the `io_uring_setup` syscall that returns a ring file descriptor, and takes a big struct with parameters.

Ring access is done through memory mapping. Reading and writing the ring works with a head and tail indexes that are common to ring data structures.

To send a new sqe, one reads the current tail, verifies if the ring won't be full, writes data just after the tail, then updates its index in two steps with write barriers between each. For cqe, it works a bit similarly.

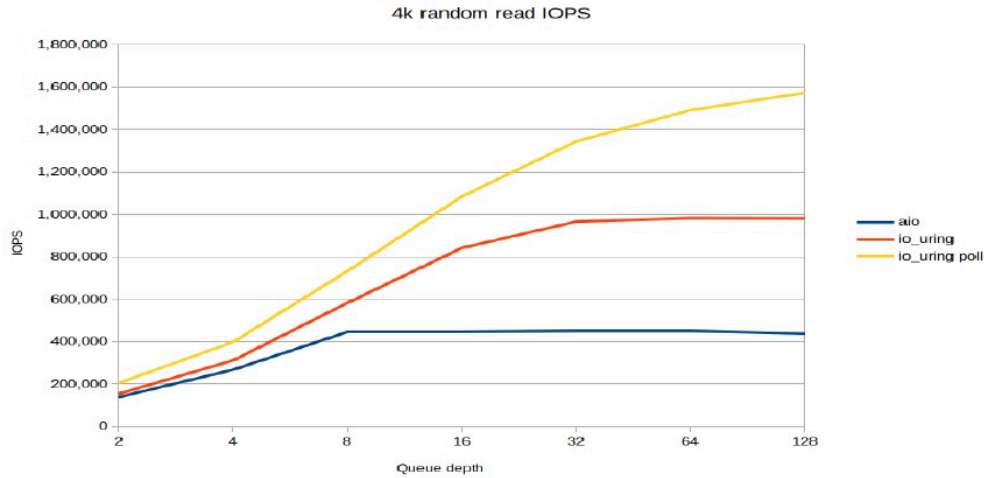
To submit requests, one uses the `io_uring_enter` syscall, which enables both submit and complete at the same time.

Jens says that if this looks a bit complex, it's because it's not made to be used directly in an app. There's `liburing` for that which handles the complexity of ring access behind the scenes.

liburing has various helpers to the multiple commands supported by io_uring (read/write/recv/send/poll/sync, etc.). It has a few features: it's possible to wait for the previous command to drain. It's possible to link (chain) multiple commands, like write->write->write->fsync. There's also an example to do direct copy of an arbitrary offset within a file to another within another file in examples/link-cp.c.

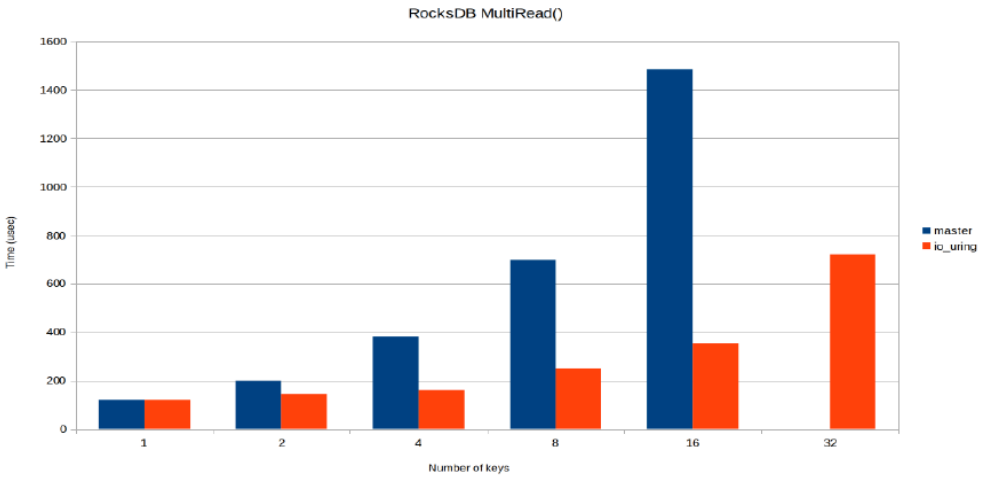
It supports multiple features based on registering aux functions for advanced features, like eventfd completion notifications.

Polled IO is useful when wants to trade CPU usage for latency. It's supported automatically by ioring at setup time, but can't be mixed with non-polled IO. It's also possible to do polled IO submission where reaping is app-polled in order to do I/O without a single syscall.



In addition to having better raw IOPS and buffered read performance, io_uring also has better latency.

Adoption is really growing, with interest in the Rust and C++ communities, in Ceph, libuv and postgresql. In RocksDB, the MultiRead() performance greatly improved:



A simple single thread echo server written by @CondyChen showed better performance than with epoll. Intel has also had great NVMe improvements on IOPS from 500k to 1-2M IOPS when using io_uring vs aio.

In the future, Jens imagines any high perfomance syscall that needs to be fully async could be implemented on top of io_uring. Jens maintains a definitive documentation of io_uring, which is being updated regularly.

The next steps toward Software Freedom for Linux — Bradley Kuhn

Bradely first downloaded Linux in 1992, and didn't boot it before 1993. Although he because a Free Software activist, he started as a computer science student who built a Linux-based lab in 1993. He had to patch the kernel to remove ctrl-alt-del to prevent any local user from rebooting the machine and disconnecting remote students.

That's when he did this small modification by using the fundamental freedom provided by the GPL which Linux used, that he understood how important it was.

The freedom the first kernel developers had to hack on their own devices led to a very useful OS. The first Linksys WRT router used Linux because of this. But it was also the first massively sold GPL-violating device. Once this violation was resolved, this lead to the creation of the OpenWRT community, which gave even more freedom to the users.

With time, more routers were released violating the GPL, which Harald Welte helped resolve, sometimes with litigation. This helped the OpenWRT project being ported to even more projects.

Bradley says that Linux is the most successful GPL software project out there. But to stay there, it needs users to be able to continue installing their modifications and tinkering.

GPL enforcement is a necessary part for that, Bradley says. Although he doesn't support malicious enforcers that want to get rich out of it. He says that Software Freedom Conservancy is available to help do ethical GPL enforcement.

That's it for today ! Conitinue with the last day live coverage.