Live Blog Day 1 – Morning

POSTED ON SEPTEMBER 26, 2018 BY ANISSE ASTIER

Welcome to the Kernel Recipes Liveblog !

This is the 7th edition of kernel-recipes, and the first year with an official liveblog.

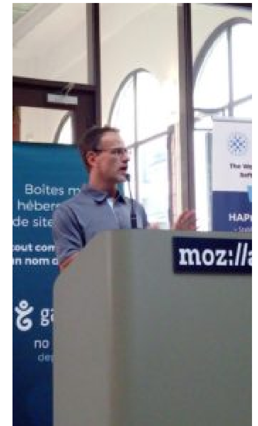
Kernel Shark 1.0 : what's new and what's coming — Steven Rostedt

Steven says that KernelShark, a visual tracing tool, is still progressing and quickly reaching 1.0 release status. They now have a full-time employee working on it at Vmware.

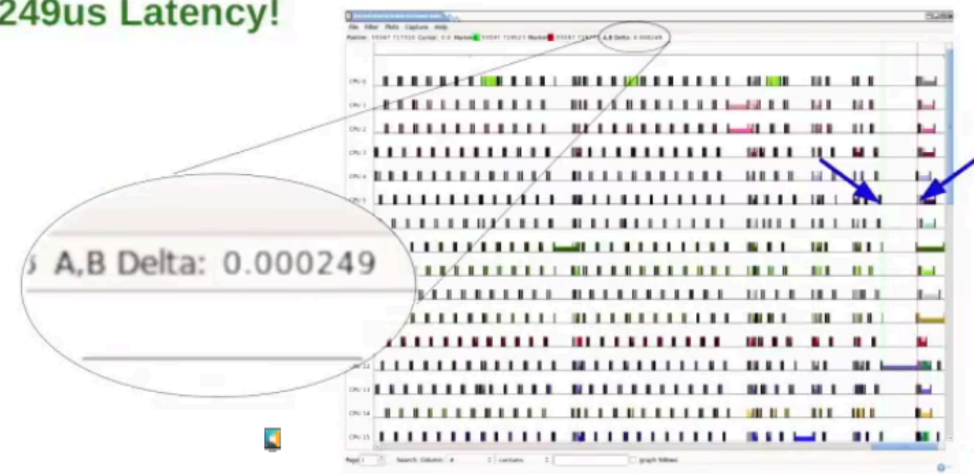
KernelShark is a visual interface to trace-cmd, a frontend for ftrace, a kernel tracing framework. Originally written with GTK+ 2.0, they needed to port it to GTK+ 3.0, but were burned by the porting effort, so they decided to move to Qt.

KernelShark was created to help visualizing text traces, which can be quite hard to understand as they grow.

Steven recounts that time when a customer couldn't detect a hardware latency, because they had tool to detect them written in Java, with a wrong window of observation. Cyclictest, Steven's tool could detect them quite easily. But still, it was very hard to understand the text traces for someone not use to ftrace; with KernelShark, everything appears instantly.



KernelShark 1.0 is much faster thanks to Yordan Karadzhov's rewrite of the core algorithms. It can display the data at a much higher rate. Steven does not claim any authorship, but has been mentoring and reviewing Yordan's code over the last months, from prototypes to the production phase, which is approaching very quickly, since there's a conference next month where they want to announce 1.0. Steven calls this "Conference-driven development".



KernelShark's 1.0 UI has a few different types of "Markers" to locate yourself in the trace : Markers A and B are used to measure differences between points in time in a trace. It's been simplified since 0.2. You can still zoom-in, but you get more dense information in the zoomed-out views. A lot of efforts has been put in making sure finding what where you're at in the trace is simpler, and making browsing a trace simpler.

What's next after 1.0 ? Steven says they want to move all the core algorithms in libkshark.so, a future LGPL library. This will allow you to write your own tracing UIs, without all the efforts of managing ftrace, parsing traces, and understanding the content. Python bindings are also in progress. The goal is to have the core engine in libkshark, that will then be used in many applications.

Future improvements planned include different kind of views (plot views, flame graphs, etc.); as well as showing traces from multiple machines.

To conclude, Steven gave a demo of KernelShark's new performance, and new UI features.

In answer to a question from the public, Steven clarified that KernelShark's goal for 1.0 was to be on feature-parity with the previous, GTK version, then add new features, like plugins.

Another question was on the binary trace format, and Steven says that it's fully documented in the trace-cmd.dat manpage.

XDP: a new programmable network layer — Jesper Dangaard Brouer

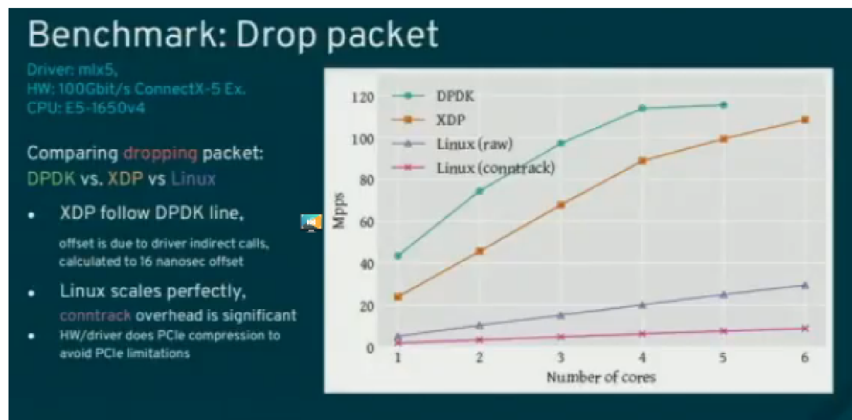
XDP stands for eXpress Data Path which arrived recently in Linux. Jesper says the goal of talk is to introduce the rationale and basic building blocks behind this upstream technology, not necessarily dive in too deep in the tech details or learn how to use eBPF.

XDP is designed as a new layer in the Linux network stack, before skb allocation. We can consider this a “competitor” to kernel bypass technologies like DPDK or netmap, but this is not a kernel bypass, Jesper says, since the dataplane is kept in the kernel.

Kernel bypass solutions market themselves as 10x faster than Linux kernel netstacks, which is true, but for their specific usecase. Unfortunately it let some people think that the kernel is inherently too slow to do networking, which pisses off Jesper, and led him to work on XDP.

An XDP goal is to close the performance gap with kernel-bypass solutions, but not necessarily to be faster; another is to provide an in-kernel alternative, that is more flexible, with hooks at the driver level that allow fast eBPF processing, and still keep using the Linux kernel network stack.

Jesper showed a benchmark, showing how many millions packet per seconds XDP can handle, and how close it has come to DPDK, and even being faster when L2 forwarding packets on the same NIC.

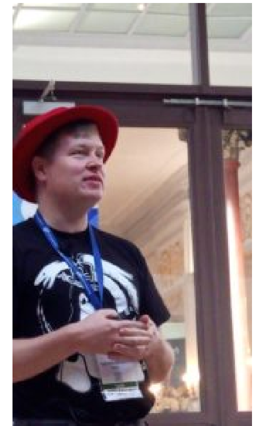


XDP works with BPF programs, which return an XDP action, to decide what to do with a packet: drop it, pass, tx, abort or redirect.

It allows cooperating with the network stack by modifying headers, for example to change the rx-handler in order to handle an unknown protocol by the kernel.

The data plane is still inside the kernel, and the control plane is loaded by userspace through the bpf system call. BPF programs are sandboxed code running inside the kernel. BPF kernel hook provide limited possible actions, while still allowing users to have programmable policies.

XDP_REDIRECT is an XDP action that allows redirecting raw frames to another device. Redirect using BPF maps is a novel feature, because it allows batching through RX bulking from driver code, into a map, delaying expensive flush operations in NIC tailptr/doorbell. Redirecting into a map also allows creating new types of redirect without modifying driver code, so Jesper is hopeful that this will be the last XDP action, which need driver support.



There a few BPF-level redirect map types: the devmap type allows redirecting te a given network device; the cpumap type allows redirecting a raw XDP frame to another CPU thread where the kernel will do skb allocation and traditional netstack handling. The xskmap, allows redirecting raw XDP frames into userspace through the AF_XDP socket type.

Spectre V2 countermeasures killed XDP performance with the retpoline indirect call tricks. It’s still processing aobut 6M packets per second per CPU core, but it used to be about 13M before. Jesper can’t just disable retpoline entirely, but there are solutions in the pipeline, which will require improving the DMA use, as shown by Christoph Hellwig in a proof of concept.

Jesper then showed how to move from NDIV to XDP. NDIV is a custom kernel solution by Willy Tarreau (presented a few years ago at Kernel Recipes), which never went upstream, but is in use in HAProxy. Jesper showed an architecture to offer all the features needed by HAProxy with XDP. Whether this would then be upstreamable or not is still to be determined though.

Jesper finished by insisting that XDP is a combined effort of many people, and thanking them.

A remark from the audience was that XDP’s integration with the kernel, allowing to inject packet in the netstack very easily is a huge advantage over other kernel bypass technologies. In response to a question, Jesper than shared an optimization trick, where reducing the size of an internal ring buffer allowed reducing the number of cache misses (as measured with perf), and then greatly improving the results.

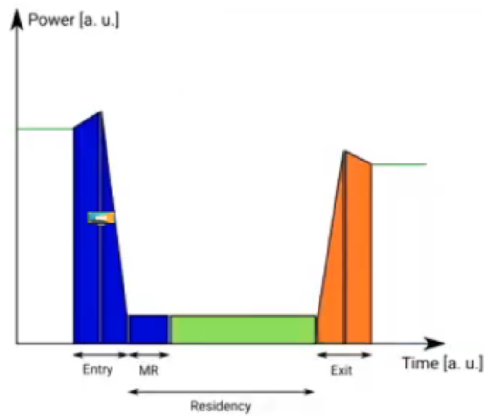
Since time permitted, Jesper continued on to explaining in more details how the cpumap redirect work. The main goal is to send a raw XDP frame to the network stack, without the impact of a deep call into the netstack, since the call happens on a different CPU, allowing it to be processed concurrently, and the small eBPF program continues working without moving out of the CPU cache. cpumap uses a few in-kernel tricks and integration with the scheduler, using kthread enqueuing, to make sure the multiple-producers single-consumer model works well.

CPU Idle loop rework — Rafael J. Wysocki

Rafael has been maintaining multiple subsystems from cpuidle to cpufreq and power management.

He started by introducing the terminology: how to differentiate a CPU core, or a Logical CPU when SMT is taken into account (HyperThreading), in this case the scheduler see them as independent core.

As defined in Linux, a CPU is idle when it has not given task to execute. It does not necessarily mean that no code is running on it. But Linux also uses this idle state to reduce the power consumption of the CPU.



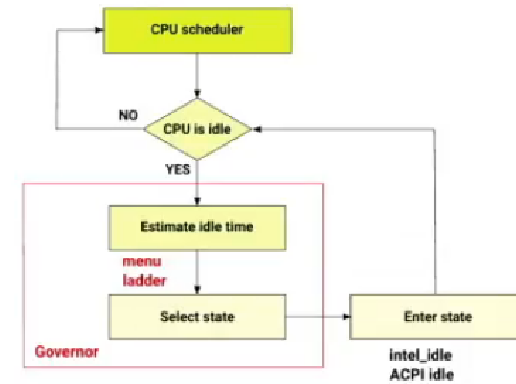
There's a fist step to prepare for lower level of power consumption: first phase (Entry), where power consumption will slightly increase. Then a similar phase when going out of lower levels (Exit). Both of these add incompressible latency, which needs to be taken into account before deciding to reduce the power level.

On Intel hardware, the idle states translate to C-states. But the C-states concern CPU cores, not necessarily Logical CPUs, so there needs to be coordination, and understanding at kernel-level of the package layout and relation between logical CPUs, and what forms a CPU core.

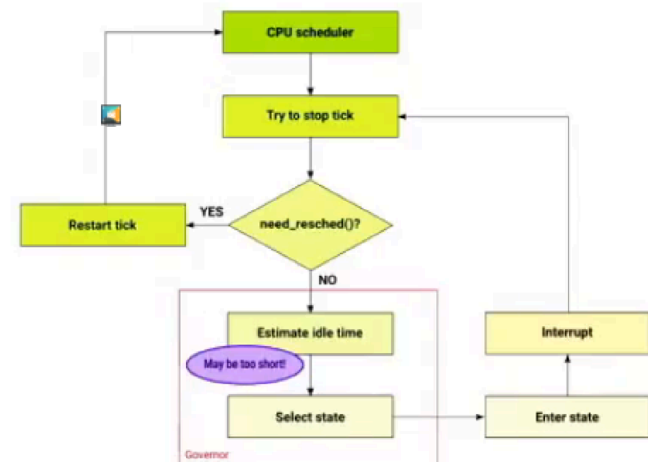
The high-level CPU idle time decision algorithm requires interaction between a few components: the CPU scheduler, which decides when a CPU should be idle. Then there are idle governors, two at the moment: menu and ladder, which estimate the idle time, and then select a state.

When entering the state, this is handled by the drivers, for example intel_idle or ACPI idle.

This high-level view looks simple enough:



But there are complications. The CPU Schedule Tick timer still needs to run regularly, but it can be stopped when there are no tasks to run, since it's going to be idle anyway. So the original design implemented is a follows, stopping the tick upfront, before deciding whether or not the idle time would short or long:

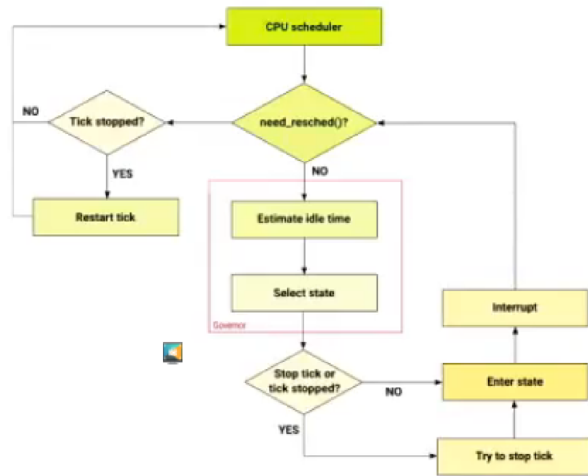


This unfortunately was not good for short idle times, since it did not necessarily need to stop the tick, and would then lead to longer-than needed idle times, and more latency. It also caused an issue with the idle governor, which handled two source of information, one of which was non-deterministic.

The "menu" Idle Governor takes information from the NOHZ infrastructure to determine the next timer event, and from the CPU scheduler utilization information, and then computes an Idle time range, which

is used in conjunction with last wakeup statistics to predict an idle duration. But the wakeup statistics might be influenced by repetitive timers, which doesn't make sense.

So this was rewritten in 4.17, where the Idle governor decides whether or not stop the scheduler tick. But the governor also needs to be changed to take into account the next timer event. So it was integrated with the NOHZ infrastructure as well.



Rafael showed results where the power consumption of Intel servers was reduced after the Idle loop redesign. But these improvements, Rafael says, were a side-effect of the idle-time improvements. These power improvements also didn't directly affect laptops, since they don't have the same type of instrumented workloads.

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

Live Blog Day 1 – Afternoon

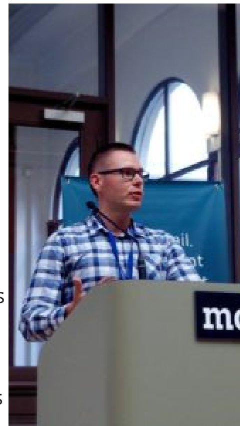
POSTED ON SEPTEMBER 26, 2018 BY ANISSE ASTIER

This follows the morning liveblog.

New GPIO interfaces for User Space — Bartosz Golaszewski

A GPIO is General-Purpose Input/Output pin at the hardware level. It can be configured at runtime, enabled disabled, it can interrupt the CPU. Within this relatively simple features, it has many applications, from buttons, to leds, buzzers, thermostats, pumps, etc.

Currently in the kernel there's a provider-consumer model with two co-existing interfaces: a legacy one, deprecated, and another one, based on GPIO descriptors, which allows more fine-grained control.


Bartosz Golaszewski

In general, Bartosz says, you should not be using GPIOs from userspace, but you might in reality often need to if you don't have a kernel driver for your devices.

The /sys/class/gpio interface is considered the legacy user API. Since it uses sysfs, state is not tied to any process, and you have concurrent access to sysfs attributes. A crashing process will not clean its exports for example. The API is also cumbersome to use, since you have multiple attributes per GPIO, polling is complex etc.

In Linux v4.8, a new character device-based ABI has been merged. It uses one device per gpiochip, with a classic open/ioctl/poll/read/close interface. It allows requesting multiple lines at once, and write multiple values at once. The polling is simpler, and the events are buffered in the kernel, so that you don't miss any.

Bartosz showed a few code example on how to use this API directly, but advised to use a new library libgpiod, he wrote at job at Baylibre, initially to toggle power switches. The library evolved, and now has its full API documented in doxygen, has bindingis for C++ and python3, many command line tools and a custom test suite. It provides Iterators, which aren't in the kernel API, as well as other higher-level actions.

The tools in the libgpiod project allow enumerating the devices, listing their features, control the gpios, monitor them, etc. It has features to fully control the GPIOs from scripts, without being cumbersome.

Bartosz then woke up the room by saying he liked C++, while talking about how much easier the libgpiod C++ bindings are to use. He even rewrote the libgpiod tools as an example for the C++ API. He did the same for the Python3 bindings, which are implemented in C as a native Python3 module.

He also started working on dbus bindings, with a daemon exposing chips and line objects on the bus.

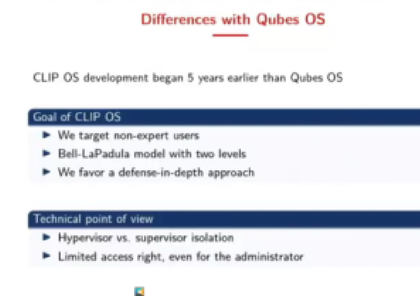
libgpiod is hosted on kernel.org, available in Yocto meta-openembedded layer, buildroot, and many distributions.

CLIP OS: Building a defense-in-depth OS around the Linux Kernel — Mickael Salaün, Timothée Ravier

CLIP OS is a new Linux distribution released last week, but it's been worked on for many years now at ANSSI.

CLIP OS is a hardened Linux kernel and userspace, without a root account. It has an "admin" account with less privileges, for example, it cannot access user data, or tamper with the system by installing an application not on a whitelist. It uses an A/B update system similar to Android and Chromeos.

The goal is to have a multilevel security OS. You have two isolated environments "low" and "high" security level; between which you can exchange data when necessary following the Bell-LaPadula model. A device (USB key, webcam, etc.) can only be accessed by one level at a time.



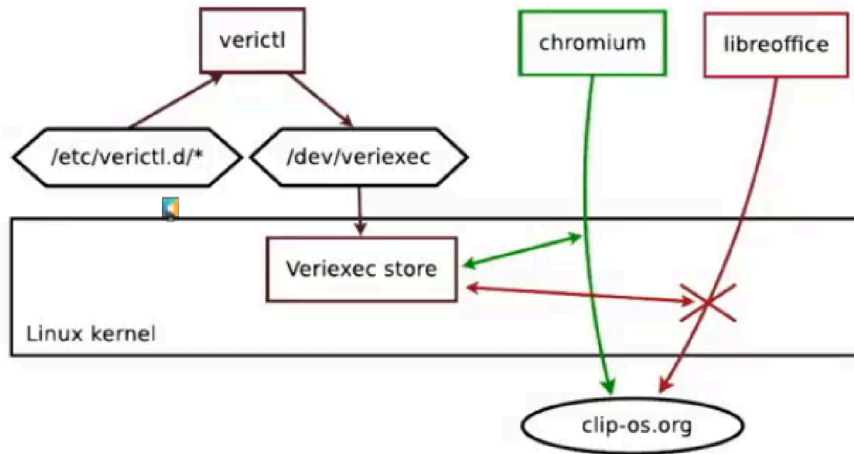
It's very similar to Qubes OS, but development started 5 years earlier than Qubes, Mickael says. The goal of CLIP OS is to target non-expert users, and favors a defense-in-depth approach within Linux. Whereas Qubes is based on supervisor and has an all-powerful admin account.

CLIP OS is based on the Gentoo hardened linux distribution. It uses the Gentoo Hardened toolchain, and the Linux-Vserver kernel patch, which adds constraints on top of linux namespaces. It's based on grsecurity and PaX for kernel self-protection and user hardening features. It also has it's own CLIP LSM to complement the linux permission model and integrate better with vserver and grsecurity.

It uses the Write XOR Execute policy for memory (through PaX), or for devices mount points. They added a new O_MAYEXEC open flag to control this at file opening for example.

They harden their containers by leveraging vserver admin and audit concepts, adding new capability bounding sets and hardening chroot.

In CLIP-LSM, they tried splitting linux capabilities by adding new permissions (verixec), which can then be bound to a given XID (for vserver namespaces). It does not use xattr, in order to be independent from the filesystem. This is configured in a file in /etc/verictl.d/



Coming in version 5

New linux kernel hardening is coming: paranoid command line parameters, and strict sysctl defaults.

The project also wants to cooperate with the upstream Kernel Self Protection project, by open sourcing, and merging in-flight hardening patches in their tree.

They do that with the linux-hardened tree, the Lockdown patch and Stackleak. The last two are closed to be merged upstream.

Landlock is planned to be merged into the CLIP OS kernel tree, but efforts are still in-progress by Mickael (its author) to submit it upstream.

In conclusion, the project is still ongoing, and looking for contributors.

Zinc: Minimal Lightweight Crypto API — Jason A. Donenfeld

Jason said that Zinc's goal is to be the most boring crypto API possible.

Jason says he's been working on Wireguard, an in-kernel VPN protocol, for the last few years. Zinc was written for Wireguard, because he found the crypto API to be too complex for a simple use case. He initially hoped to move to the linux crypto API later.

He starts by citing the big_key kernel API which was improperly designed, and had many issues. So he went on to try to fix that, and in the process learn how to use the Linux crypto api. He found that the API was fully async, had many memory allocations or incoherent designs. It was designed with crypto accelerators in mind, and not with the goal of having a simple, easy-to-review design.

big_key needed to do big 1M allocation, but some systems can't kmalloc too much at a time. Instead of using kvalloc, a solution was designed with custom allocator with direct page allocation. Jason says that while it was designed by intelligent people, it still has this Entreprise-y feeling, that didn't appeal him to use it.

On the contrary, Zinc has a much simpler design, with pure-software usage (no accelerators possible), evrything being done on CPU. It includes the chacha20 stream cipher, poly1305 one-time authenticator, an AEAD based on the two, blake2s and curve25519 symmetric and asymmetric ciphers. The goal is not be innovative, Jason says.

Dynamic dispatch can be implemented on top of Zinc. Existing code can be refactored to use Zinc. And in fact, many crypto functions have leaked into lib/, so this is what the developers seem to want. The goal of Zinc is to provide them in coherent library.

Zinc's goal is to include a formally verified implementation, when it's available. It also has SIMD-accelerated versions of many algorithms. For the formally verified versions, it uses HACL* and fiat-crypto code, which is machine-generated C code from complex formally verified models in Coq. The HACL* team is based out of Inria here in Paris. Jason wants to make Zinc easy to approach by academia, and in general professional cryptographers, whom, he says, aren't used to work with Linux kernel.

The file organization is also different: all architecture-specific implementations are in the same directory as the generic version. It then uses compiler-inlining and not function pointers, which makes it faster without retpoline spectre mitigations.

Zinc also uses SIMD-batching, to only save FPU/SIMD context at the beginning and end of loop. It has dispositions to keep the code preemptible between loop iterations if necessary.

The Zinc patchset is posted along with WireGuard, currently in v6 form. The current goal is to have it merged for Linux 5.0/4.20.

Democracy requires free software — Matthias Kirschner

Matthias started by giving an example from the french revolution: how the architecture of Paris moved from small alleys to big boulevard changed the power balance by allowing the military to move in Paris. He gave other examples of advances that changed the power balance, from reading and writing, to mathematics, agriculture, and finally to computers.

Who controls the software running in the computers, cars, planes, washing machines, pacemakers; controls the devices. Processes are also more and more controlled by software, like the admission process to multiple universities in France. So a student association requested access to the source code of this software. Then only, the admission rules could be understood. The proximity rule for instance

was discovered, and challenged because it meant that universities in Paris were more likely to accept students that lived closer, in high-rent areas, which was unfair to poorer students.

A similar case is the CV processing algorithm of some companies in the US to decide what applicant can be invited for an interview. It was supposed to be discriminating against black applicants because of the address in the application.

He then cited another software used in the justice system in some US states to guess the probability of repeating offenses, which would then influence the decision, without any recourse since the algorithm is secret.

Matthias' main argument, is that understanding the rules of our world is fundamental. If these rules are encoded in software and you can't understand them, then you can't predict what will happen or what to do.

Who has power in a world where software is pervasive is critical, Matthias says. And Free Software increases distribution of power.

It's important to provide resources and support to people, who don't necessarily understand technology, but are directly impacted by the obscurity of proprietary-software-driven decisions. The goal is to remove barriers to understanding how the software around us works.

When governments wants to develop software, they should prefer free software.

FSFE worked on reuse.software a set of best practices when releasing free software. He also mentioned the recent Article 13 debates in the EU, as well as the Public Money, Public Code campaign.

That's it for today ! Follow the thursday morning liveblog !

Live Blog Day 2 – Morning

POSTED ON SEPTEMBER 27, 2018 BY ANISSE ASTIER

Welcome to the Kernel Recipes Liveblog ! This follows yesterday afternoon's liveblog.

Atomic explosion: evolution of relaxed concurrency primitives — Will Deacon

Will is the co-maintainer of the arm64 architecture, and recently got involved on the concurrency primitives in the arm kernel. He started by saying he doesn't really like concurrency, since it's often not synonym of performance.

There are many low-level concurrency primitives in the kernel, but he started with describing atomics, operations that are guaranteed to be indivisible. Core code provides a C implementation that is generic with lock primitives. But Will says you don't want to look at this code, nor use it, for performance reasons.

Historically, atomic_t isn't well defined, separate from cmpxchg, and specific per-architecture. He's recently been working or improving this. For example, core code will generate code the architectures don't provide, using the lower-level primitives that are available.

The new _relaxed semantic extension allows having unordered code, that the compiler can reorder. But, while it's very useful, he says it's still under-used, which is how he's presenting today, so people can understand the new semantics.

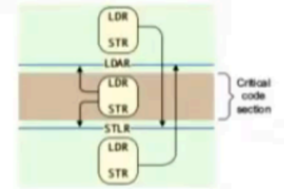
There's also a the fully-ordered smp_mb() atomic_t extension, that works across all CPUs, that is very useful, but quite expensive to use, on all available architectures.

Acquire/release is a middle ground between relaxed and full-ordered; they can be chained together without loss of cumulativity.

Acquire/Release

Middle-ground between relaxed and fully-ordered:

- Appeals to "message-passing" idiom
- Producer thread writes/releases data
- Consumer thread reads/acquires the same data
- Maps efficiently to existing architectures and C/C11
- 'Roach-motel' semantics



Everthing before a release is visible to everything after an acquire that reads from the release.

More flexible than `smp_wmb()/smp_rmb()` but without enforcing ST->LD ordering of `smp_mb()`.

13 © 2018 Arm Limited arm

Depending on the architecture, the atomic acquire or release might be identical; but this granularity allows architecture that support some semantics to implement and use them.

Generic locking

The Linux kernel has generic implementations of many locking primitives in kernel/locking/*. These are very useful, portable, and can be formally verified. But are there performant enough ?

He started by looking at qrwlock, which uses architecture-specific optimization for the rwlock semantics; qrwlock has a faster write performance, but lower read rates than rwlock.

qspinlock is also a generic implementation relying on very simple architecture locking primitives, and is very performant. Will says that it's as simple to enable for an architecture as enabling it in Kconfig. Of course, you must first verify that your memory model matches what is expected by this algorithm. He was a bit hesitant to enable it at first, because he wasn't sure it would work as expected.

So he started looking at verification tools, and testing. tools/memory-model for example, and other formal modeling tools like TLA+, a formal specification language, or testing with herdtools. But this is still quite complex, as "people are getting PhDs out of this". It's slow, but it will get there.

He's also encouraged the audience to review attentively code using atomics and other generic locking primitives, and not hesitate to Cc: him and other atomic maintainers for help in the reviewing process.

Coccinelle: 10 years of automated evolution in the Linux kernel — Julia Lawall

The Linux kernel is a large codebase, that is still rapidly evolving. But it poses an issue on how to do at-scale evolution of kernel code. A simple idea is to raise the level of abstraction to write semantic patches. That's were Coccinelle comes in with it's SmPL Semantic Patch Language, and tool to apply these modifications. It was first released in 2008.

There were over 5500 commits in that period of time. It started originally with just Coccinelle developers, and then started to be used by other kernel developers, including recently outreachy interns. Julia noted that 44% of the 88 kernel developers that have a patch that touches at least 100 files have at least one commit that uses Coccinelle.

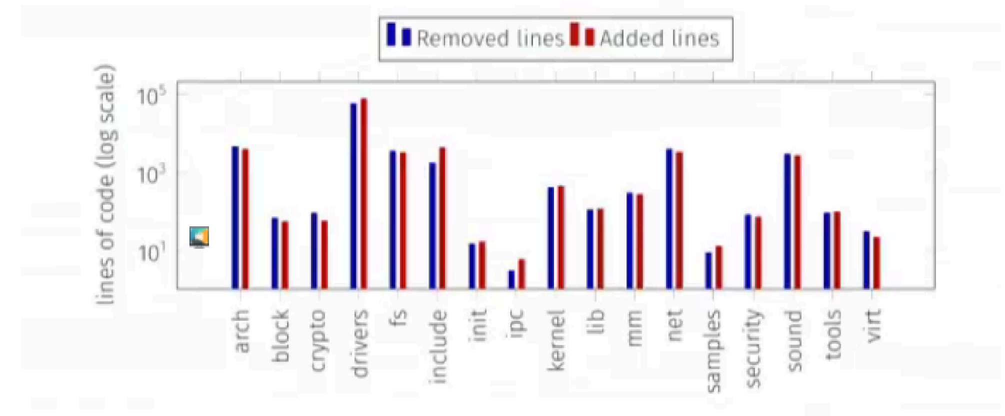
The original idea was to provide as simple as possible expressivity, so that it's easy and convenient to use. But real world constraints like language evolution also pushed them to add new features: position variables, and script language rules. Script language rules can for example print a message based on Coccinelle input (including position variables for the line of the code). Many (43%) patches made use of these new features, as well as all semantic scripts present in the Linux source tree.

An original design goal of Coccinnelle was performance: it only parses .c files, works only inside a function, doesn't include headers by default, or expand macro. But the real world constraints showed that even with this, parsing 1M, or 15M lines of code is still very slow. Parsing is slow because of backtracking heuristics. They added parallelism with parmap, and pre-indexing to improve on this. Running Coccinelle with the 59 semantic patches present in the Linux kernel takes about the same time as building the full kernel, Julia says.

Design also included correctness guarantees at its core. It's been proven to be pretty successful in that regard, as long as developers write correct semantic patches.

The strategy used by the Coccinelle team to improve their visibility was to show by example how it could be used, by solving real world issues the Linux kernel developers encountered. They found parsing bugs in unreachable code (hidden behind ifdefs), fixed many issues at refactoring points like irq, etc. This showed initial interest, and it was presented at many conferences (including Kernel Recipes !).

It's impact has reached all over the kernel; and it has been used for both cleanups and bug fixes. Julia showed how maintainers really took ownership of the tool, writing very long semantic patches without help from the Coccinelle team.



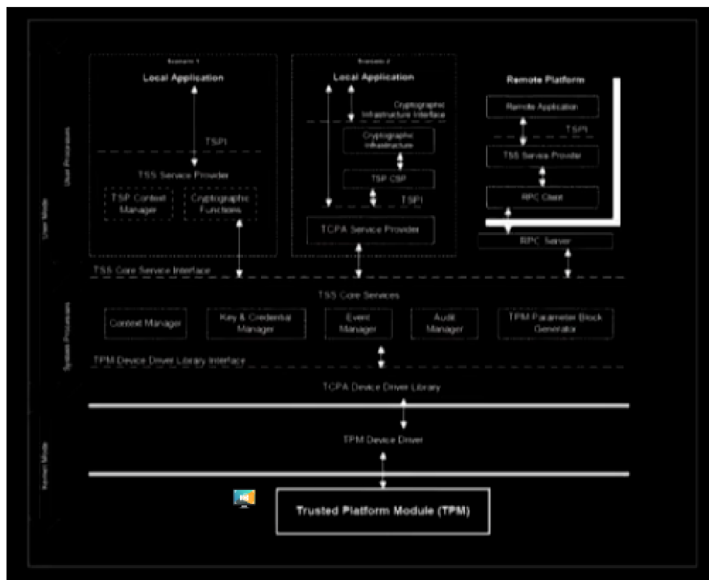
JMake is a tool built on top of Coccinelle to help verify semantic patches by making sure it builds the lines that are changed with the proper configuration. Prequel is another tool that was built on top, that helps searching the git commit history with a semantic patch, ordering them from most pertinent to less. Prequel is a frontend to Coccinelle, and Steven from the audience said it might be very useful to have git integration for this tool.

In conclusion, Julia said that the tool's initial decision are still valid, with some extensions. They kept the core, without feature creep, and had a wide impact on code everyone is running.

TPM enabling the Crypto ecosystem for enhanced security — James Bottomley

Everybody needs help protectling secrets, James started. A secret in this case might be RSA/ECC keys for identity for example. These keys, when used as identity representative, need to be protected as best as possible. Which means using a Hardware Security Module (HSM) USB key. The issue is that most of those can only carry one key.

One key is far from enough: James says he is currently using 12. A TPM can help scale beyond these HSMs limitations. TPMs are chips with shielded memory, that are now ubiquitous and present in most laptops. Unfortunately, they have a very bad programming experience. They use the TSS model, which is very complex.



Linux TSS 1.2 is badly designed with tcsd being a single daemon that is shared between all users. For TPM 2.0, it can be improved upon.

TPMs can do many things: attestation, data sealing, etc. But James focuses on Key Shielding, which key storage in the TPM. In TPM 2.0, you have algorithm agility, while TPM 1.2 is limited to sha1/rsa2048. TPM 2.0 chips usually support a few elliptic curves algorithms and sha2, the latter being a much better choice as a hash function.

TPM 2.0 generates keys from random numbers with key derivation functions; RSA keys are very slow to be generated though. James showed that it took 43s on his laptop to generate a key. It's much faster for elliptic curves, or on last generations TPMs.

You keep the seed, and only the TPM inside a device will be able to generate a key. Which means that a stolen seed is unusable without the TPM inside the laptop. Since it can never be extracted, it means that you can't really tie your identity to the lifetime of the device. If you store a key inside a TPM, you need a way to find the original back, and another way to convert it in form usable for a new TPM.

James is very critical of the TCG: the TPM 2.0 is still not complete, even if there already devices shipping.

Support is improving in Linux, with TPM 2.0 resource manager support landed in 4.12.

James says that the IBM TSS (Trusted Secure Stack) has been available for a long time and can be used to established an encrypted session from your application, directly to the TPM2 device. It's simpler to use, and based on TPM2 commands. James says you can build a secure crypto system with only 5 commands. It's possible to generate the key directly on the TPM, but since the Infineon prime bug, most people don't trust TPMs anymore, and generate them outside the TPMs before importing them. The keys aren't really stored inside the TPM, but TPM provides an encrypted blob, which the app must store, and reload when it needs to use. There are other functions for signing, and rsa/ecdh decrypt.

Since there's a lot of sensitive information in the commands, there must be a secure channel to the TPM, with the ESAPI, an encrypt session api. This ensures nothing between the TPM and the app can get the data that is exchanged. When the app is the kernel (for random number generation or storage), direct snooping on the bus is potential attack model thwarted by ESAPI.

An issue with the ESAPI is that only the first parameter (!) is encrypted. So they had to add new commands to take this into account with double-encryption of the other parameters.

A disadvantage of TPMs is that keys are tied to a single physical laptop, and need to be re-converted when you change laptops. You also need a kernel more recent than 4.12 for TPM 2.0.

Integration with OpenSSL with an external engine is progressing well on James' side, and works well. An issue is that not all Elliptic Curves are supported, the only two mandated being BN-256 and P-256. Curve 25519 is not even on the TCG radar. It's possible to load openssl, openvpn, openssh, and more recently gnupg keys. James showed a demo of generating and loading gpg key into a TPM, and then protecting a new subkey with the TPM.

James says this works really well and is integrated with many software projects.

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

Live Blog Day 2 – Afternoon

POSTED ON SEPTEMBER 27, 2018 BY ANISSE ASTIER

Welcome to the Kernel Recipes Liveblog ! This follows the morning liveblog.

From knowing the definition of Linux kernel to becoming a kernel hacker — Vaishali Thakkar

Vaishali want to recount how she became a kernel hacker. She initially thought that to be kernel programmer, you had to be a wizard C programmer.

Unfortunately, she learned about programming at university using Turbo C, an environment very different to Linux+gcc. The linux kernel was described very briefly, and Unix commands were learned on paper.

Last but not least, she said english is not her mother-tongue, and she learned it pretty late so this was another barrier.

A very bad thing then happened to here: she got a leg fracture. But this was the starting point of her Linux journey. Freed of assignments, she could install a Linux distro on her computer, and take time to read a Linux book. Her discovery of GCC was mind-opening, and she decided to re-learn C programming.

Shortly after, she found the outreachy project, with maintainers giving specific tasks. She was accepted, which then helped her find an internship, and then a job, before she started doing freelancing.

Then she asks: do you really need to get a fracture to become a kernel hacker ? Of course not. It's important to have a growth mindset; which means that you're always ready to learn and understand your blind spots so you can improve. But is a positive attitude just enough to become a kernel hacker ? Again, she says it's not sufficient. You need to put in the efforts, and you also need to learn self learning skills.

Knowing how to find the right resources: from source code, to LWN articles, to mailing list archives, to git history/blame, to going to conference talks like Kernel Recipes.

Asking questions is also important, especially from Vaishali's background where it wasn't really celebrated. And you need to understand the difference between a good and bad question, as well as the sub-project mailing list etiquette. Various maintainers have various preferred style of approaching issues and patch submission, which is something you must understand.

Automating the learning is also important. Using tools to augment the speed at which you "git grep", from inside you editor for example. Writing your own support scripts is also very useful to reduce repetitive patterns.

You also need to keep your kernel knowledge updated: APIs evolve, submissions processes, and you need to understand this.

Last but not least, improving your general programming skills: understanding of what is intelligent or bad code, learning from other developers.

In response to a question from the audience, on what could be improved to the onboarding process of kernel developers, she said that a proper TODO would go a long way to help find new tasks, and she's maintaining her own TODO list for newcomers.

Live (Kernel) Patching: status quo and status futurus — Jiri Kosina

Why is it interesting to live-patch a kernel instead of rebooting ? Downtime has a huge hourly cost, more than $100K for many enterprises, Jiri says. Live kernel patching is for changes that are outside of the planned maintenance window, when you need to act quickly.

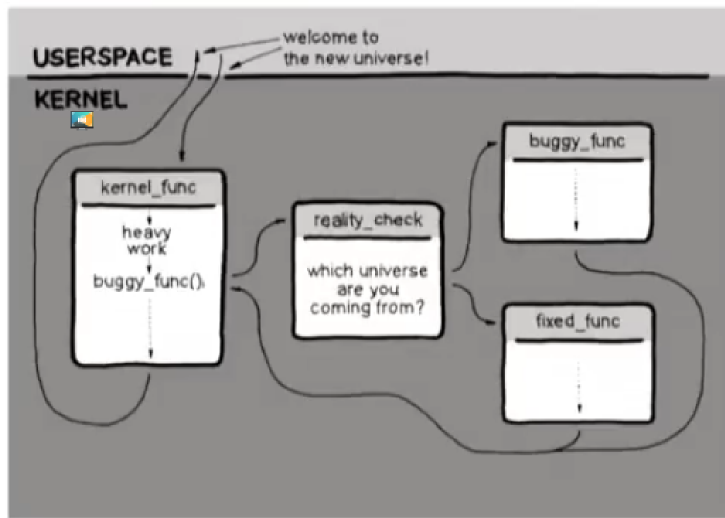
History

It all started in the 1940s, with punch-card based computers, where differently-colored punch cards would be replaced at runtime to fix bugs without downtime. In 2008, ksplice was started originally as research project to live-patch Linux, then acquired by Oracle in 2011, and source closed before commercial deployment.

In 2014 an interesting collision happened: kPatch and kGraft were released a few weeks apart, the first one from Redhat, and the second one SUSE. Both were commercially deployed

CRIU (Checkpoint/Restart in Userspace) was also another solution circa 2008, but it wasn't really immediate since it required hardware reset.

This is how kgraft/kpatch work, simplified, by using a trampoline to replace buggy_func with fixed_func:

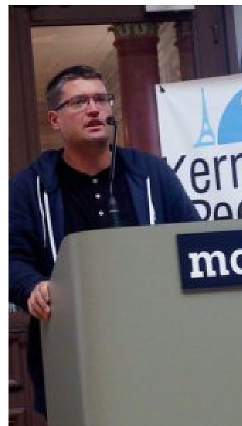


In 2015, Redhat and Suse agreed to cherry-pick individual features of each of their patchset and “merge” there approaches, with the core being upstream.

Live patching was initially x86-only, and now works on s390, ppc64 and arm64.

With the upstream livepatch infrastructure, patch generation is mostly done by hand, Jiri says. It comes in the form of a C file, that will be loaded as a kernel module.

Limitations and missing features



Currently, there’s a limited ability to deal with a data structure change, or semantic changes. The new functions need to understand both the old and new data format. Transforming data structures on access is possible, but only after lazy migration is complete.

Shadow variables is a new feature that was merged, that allows associating new fields to an existing structure, which can then be used by patched callers.

When system state contains locking mechanisms, it’s almost impossible to livepatch without stopping the system.

Trying to know when a change is or isn’t within the consistency model is very difficult. For example, you can’t really patch schedule(), or static variables in a function scope.

There’s some tooling, with a patched GCC, to find when a function has been inlined, and then fixing all the function sites where it is included.

Kprobe transferring is also missing, since there’s no match between the old function, and the patched version yet. Steven Rostedt in the room says that it could be fixable.

Extending to new architectures needs support for FTRACE_WITH_REGS, as well as a reliable stack unwinding.

There’s the inability to patch hand-written assembly, because it’s not ftrace aware, which the livepatching infrastructure uses. It’s possible in theory, but complex. Userspace patching is something that could be done, but is a different problem entirely: there’s no consistency definition, and while the kernel (and livepatch) heavily rely on GCC, it’s not the same in userspace. There are a few projects that attempt to do that, though.

Jiri then shared exclusively that SUSE is currently working on a way to patch only the system libraries by tracking the boundaries (PLT entry/trampoline return), which should be published soon.

CPU Isolation — Frédéric Weisbecker

What is CPU isolation ? It’s when you want to the full processing power for your use, without interference from other CPUs.

Kernel isolation is hard. It’s made of tradeoffs, and often requires sacrificing something, Frédéric says.

Task affinity can be used to assign a task to a CPU. For kernel threads, you can do the same sched_setaffinity(), but unbound workqueues are a special case. Unfortunately, per-CPU kThreads can’t be affined.

IRQ affinity is possible as well, but requires understanding the correct level at which to affine them.

For time IRQs, there’s no general rule, and you need to understand the timer associated with your IRQ, Frédéric says. For example, with the watchdog hrtimer, you can disable it partially with a cpumask, or entirely, although this isn’t advised. For the clocksource_watchdog, Frédéric says, the only solution is to change your hardware. Or use the black magic tsc=reliable boot option, which shouldn’t be used in production.

Nowadays, many Linux distributions enable CONFIG_NO_HZ_FULL at compile time, which disables the tick timer entirely; but you need to pass a kernel command line argument to enable it. Frédéric says it does not come for free, as there’s still overhead: you still have tick reprogramming on IRQ exit and context switch. There’s also overhead on kernel entry/exit (syscalls, exceptions, faults) for cpu time accounting, since you don’t have a tick anymore to maintain it.

Nohz requires running only one task at a time, and avoiding kernel entries entirely in order not to affect per-CPU async code in the kernel (kthreads, timers, workqueues, RCU, etc.).

Nohz housekeeping is “the lamb to be sacrificed”, Frédéric says. You still have a 1Hz CPU 0 tick, for timekeeping for example.

To have proper userspace CPU isolation, you need to avoid syscalls entirely. Some extreme cases reimplement kernel features in userspace, like TCP stack for example.

To do all that, Frédéric says it would be nice to have a unified interface to do this. He tried to extend isolcpus= to replace nohz_full= he said, but it might be better to use cpusets.

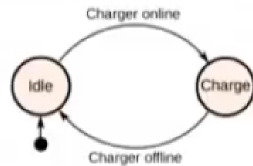
The power supply subsystem — Sebastian Reichel

The power-supply subsystem includes smart batteries, chargers, and currently lives in drivers/power/supply. Sebastian took over the maintainership of the subsystem when he was trying to submit a patch, and the maintainer wasn't responding and had disappeared.

A recent modification, he said, was big improvements by Adam Thomson to the documentation. Adam also added the usb_type property, which shows all supported modes, and allows changing mode.

Another addition, by Liam Breck, was support for dumb battery description, which don't have their own smart charging circuits.

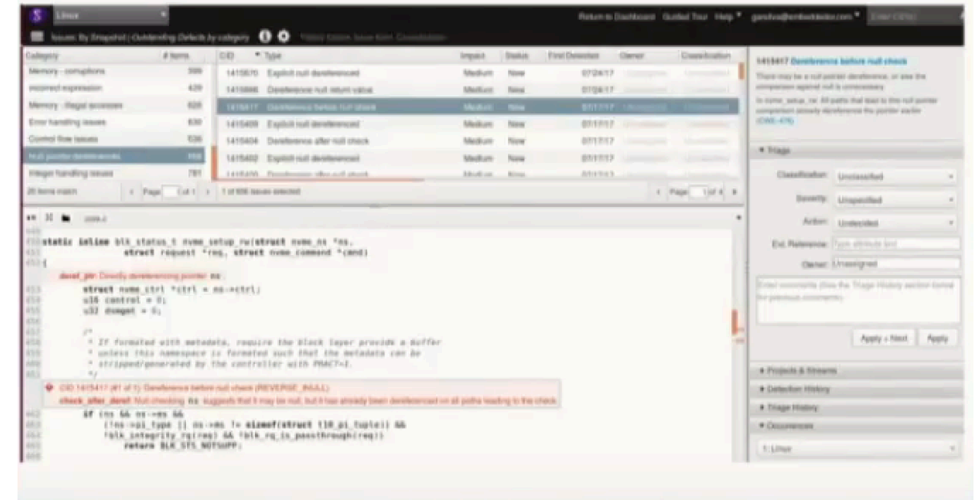
A current limitation, is the missing support for batteries that have multiple fuel-gauges, since only one is exposed to userspace; there is still no solution for this.



The current framework expects the chargers to work autonomously, as is done on x86 by the EC controller, but this is not true in embedded devices, which might need handling from kernel.

A year of fixing Coverity issues all over the kernel — Gustavo A. R. Silva

Coverity is a static code analyzer, that analyzes the code, but generates a lot of false positives. Gustavo says he spens a lot of time trying to understand what constitutes an actual bug. The high-impact issues include out-of-bound accesses, resource leaks or non-initialized variables, and medium-impact include NULL-dereference or control flow issues, for example.



He showed different examples of buggy code, which he fixed in the kernel. Incorrect variable type (size), unreachable code, an infinite loop which led to out-of-bounds access, integer overflows or user-after-free errors.

As part of his workflow, Gustavo reviews daily Coverity reports. He started recently using Smatch to find spectre v1 issues. He uses a lot Coccinelle (presented this morning at kernel recipes): writing and running scripts. He also modified his sleep schedule to wake-up earlier, in order to fix issues faster when the daily Coverity reports arrive, as part as a friendly competition with a Canonical kernel developer.

Gustavo says he's submitted 750+ patches since last the Kernel Recipes 2017 edition, up from ~250 patches the year before. About 30% of the total contributions include intentional fall-through annotation of switch cases. He reaffirmed that he really loved Coccinelle, with another 30% of his contributions coming from the tool help.

His work impacted about 939 files, with 1126 email interactions in general. Of those exchanges, he got 3 emails that stood out : "This crap…", "I hate when…". He says that's about 0.27% of the total interaction, so kernel development is 99.73% of pleasure 🙂 ; Gustavo added that he is satisfied with kernel development in general.

Once Gustavo was finished, his talk triggered a discussion on compiler warnings in the room, and on the way they should be enabled or not. Many ideas were proposed, but no conclusion was reached.

That's it for today ! Continue with the last day morning liveblog.

Live Blog Day 3 – Morning

POSTED ON SEPTEMBER 28, 2018 BY ANISSE ASTIER

Welcome to the Kernel Recipes Liveblog ! This follows yesterday afternoon's liveblog.

Meltdown and spectre: seeing through the magician tricks — Paolo Bonzini

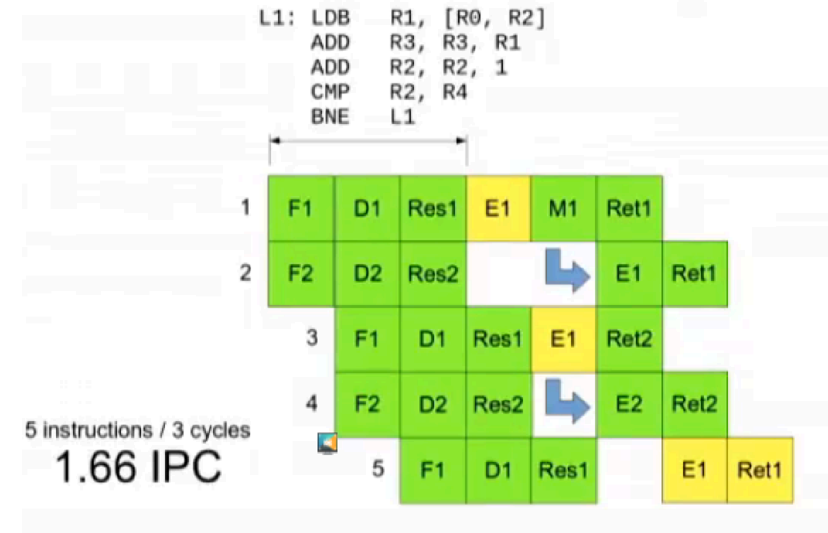
A commonality of modern computers is privilege separation. The modern OSes rely on the processor hardware for help in order to do this. The main idea behind Spectre and Meltdown is that the OS can't do that if the hardware doesn't help.

Modern CPUs usually execute instruction within a pipeline, which every phase being done in parallel (Fetch, Decode, Execute, Writeback is a simple pipeline modelization). The issue when executing conditional branches is that you never know which side of a branch will be executed, which means you might have to block the pipeline until you have the information. Or you can use branch prediction, and try to guess which condition will be correct. If the guess is wrong, the pipeline needs to be flushed.

Static branch prediction already gives you very good results, between 70% and 80% correct guesses, Paolo says. But modern CPUs use dynamic branch prediction, with more and more predicting techniques: store bias (1977), store history (1991), multilevel, neural network predictors, geometric predictors; it's a very wide field. The goal is always to improve the pipelining, to try to predict as many correct branches as possible.

Another CPU trick, is superscalar execution and scheduling. Which means that you can execute more than one instruction per cycle. It needs proper ordering of instructions though, and the compiler help, like in the infamous Itanium architecture.

Out-of-order execution is another trick that is used to improve execution speed, that is done in-CPU. It reorders instructions to execute what's possible to execute at a given stage in the pipeline, while it waits for memory fetching, or resolution of other instructions. Once all previous instructions are executed, an instruction can "retire", which is another stage in a pipeline. It's more efficient than the superscalar mode, Paolo says.



Speculative execution can then be added on top. The processor will continue executing more instructions before knowing if an instruction can retire. It greatly improves speed upon correct branch prediction, and in case of flush, it's as slow if the CPU had waited for an instruction to retire.

Caches add yet another layer of complexity: they are pervasively used because they help with memory access, which is much slower than CPUs. Cached data or instructions are much faster to fetch than in memory. The issue is that you can have cache side-channel attacks, by measuring the time it took to fetch a bit of memory, in order to know whether it was in cache or not.

Spectre

The idea behind a Spectre is that you find a memory load with an address the attacker controls. You then train the branch predictor to execute the memory load speculatively. Then you need a second access with an address that depends on the first value you read. Measuring the cache access time after that lets you read leaked data.

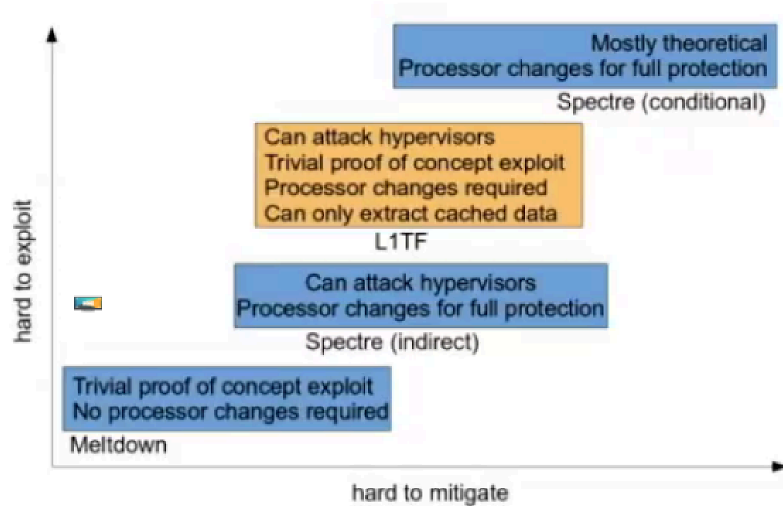
So how to work around this ? It can be done in source or at the compiler level. This is mostly done by hand now, Paolo says, at privilege boundaries.

Meltdown

Meltdown relies on paging: each program has a different memory mapping, and page faults happen when trying to access a memory address it doesn't have access to. Meltdown relies on speculative execution behind page faults, meaning you can observe cache-side effects of speculatively-executed instructions on a memory address the program doesn't have access to.

A work around this is to separate page tables between supervisor mode (kernel) and the user processes. Which means the kernel is no longer mapped user programs, but you need to do expensive TLB flush at privileges boundaries.

Another thing that can be done, is to add checks to the cache and hide the content of cache lines if the permission checks fail. This change needs new hardware and updated microcode is not enough to fix Meltdown.



To conclude, Paolo said that hardware help to mitigate these attacks will arrive, in time. And not to panic, because these are neither the first nor the last hardware bugs.

Mitigating Spectre and Meltdown (and L1TF) — David Woodhouse

A speculative execution attack, David says, has three components. First, a way to enter speculative execution: a conditional branch, an indirect branch, an exception, or TSX (Intel transactional memory extensions). Then, you need a way to prolonge the speculative the execution: with a data load, dependent loads, or dependent arithmetic. Finally, you need a way to leak data: through the data cache, which was seen in the wild or others that are theoretically possible: the instruction cache, the prediction cache or the translation cache.

Meltdown allows direct read from non-permitted (kernel) memory. It runs entirely in userspace, which means the kernel can't notice anything happening. It works on Intel (not AMD), POWER, and ARM Cortex-A75, David says.

Spectre v2 is based on indirect branches, and v1 on conditional branches. The userspace attacking the kernel needs to have two loads in a row inside the kernel, with the first one being from an address controlled by userspace.

To mitigate Meltdown, KPTI/Kaiser relies on changing the page table when going into the kernel. Which means each process has two sets of page tables.

Spectre Mitigations

Spectre v2 mitigation relies on microcode features. There are new functions in MSRs (Model-specific registers): IBRS and IBPB which allows restricting speculation, and flush the branch prediction. Linux still doesn't use these new expensive instructions

What's currently to mitigate spectre v2 used are retpolines, a way to fool the branch predictor. Any indirect branch (when jumping on an address), when using function pointers will cause a full branch predictor flush and a "pipeline stall" David says, which has a huge performance impact, since Linux relies a lot on function pointers.

To "get the performance back", the solution is to call functions directly, or to inline functions which take callback functions as arguments. Inlining an iterator function can help the compiler emit a direct call, and get much better performance in loops.

Even with all its problems retpoline is still much more performant than the IBRS instructions.

The return stack buffer also needs to be cleared, because modern Intel CPUs (Skylake onwards), prediction happens on the ret instruction as well, so it's cleared with a trick on VMEXIT and context switch.

For Spectre v1, the new array_index_nospec() adds data dependency in bounds checking, and needs to be added manually at the proper places, like get_user() David says.

L1 Terminal Fault (L1TF)

This attack, announced last months, allows a VM guest, to ready any memory in a sibling hyper-thread CPU, as long as it's loaded in the L1 cache. It relies on controling the PTEs in the VM guest, on which the CPU will execute speculative instructions.

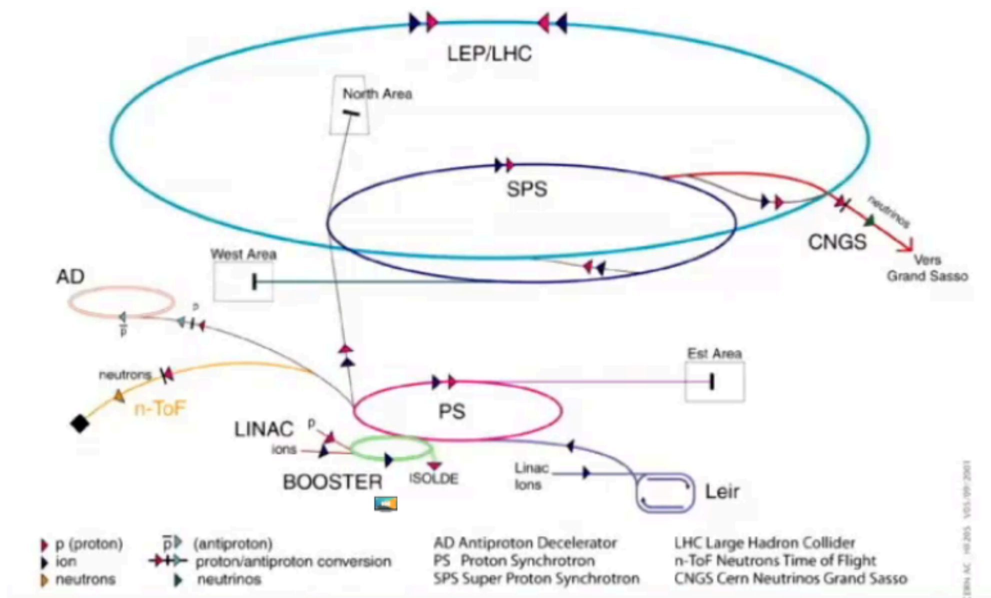
To work around this, a patchset has been posted to prevent running different VM guests on sibling cores. It's a lot of work, but there's a big motivation to do it now, David says.

In conclusion, on Linux today, there are many mitigations: retpolines, IBRS on firmware calls, IBPB on VM context switches, RSB is cleared, dcache is cleared on kernel exit. Unfortunately, on Skylake and later, there are still unsolved speculations, in many places. Xen is better, David says, since it cleares IBRS and IBPB when needed, and you don't need to "pray", when using a recent CPU.

ObsBox: a Linux-based real-time system for LHC beam monitoring — Miguel Ojeda

Miguel works for CERN, as a software engineer, and is the second person on stage this year to admit liking C++.

At CERN, they work on particle physics, which is really easy, Miguel says. It only has 9 simple steps: first, you accelerate the particules, then smash them in opposite directions. You detect the collisions byproducts, filter the in real-time huge amounts of data, store what is deemed important losing everything else. You repeat this, and then analyse to the data to test if the the expectations match reality. Miguel says the last step is to get a Nobel prize.



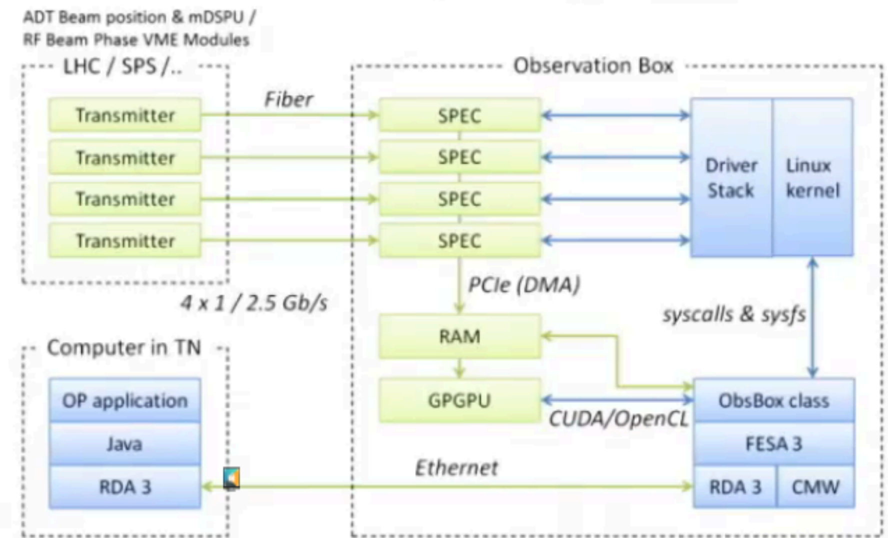
The LHC is the last step in a chain of particle accelerators, all starting in a simple bottle of hydrogen. A single proton is accelerated to just 11km/h below the speed of light.

A collision happens every 25ns, and there many type of detectors to measure the effects of this event. All the data can't be stored, so it has to be filtered in real time.

This data is then analyzed, and reused in simulations, in order to infer, with a certain degree of confidence, what happened.

To make all of this work, this requires a lot of computer science. There's a lot of C++, and a lot of Python. They regularly find compiler bugs for example. Miguel helped CERN move from CVS to git, for the collision detection software (CMS).

ObsBox is Linux system that is used to capture data out of the detections systems. Both the hardware and software was designed in-house at CERN.



They use the Linux with real-time patches. The system was relatively successful inside CERN, so that other teams were interested in doing the same. Using off-the-shelf hardware (FPGAs, and rackable Supermicro linux servers), greatly reduced the cost compared to high-end oscilloscopes.

The driver Miguel wrote is as simple as possible, he says, only needing to support read() operations for acquisitions, the rest being doing in userspace.

In conclusion, Miguel says that Linux is a great match for CERN needs because it's freely available, and easy to customize and extend.

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

Live Blog Day 3 – Afternoon

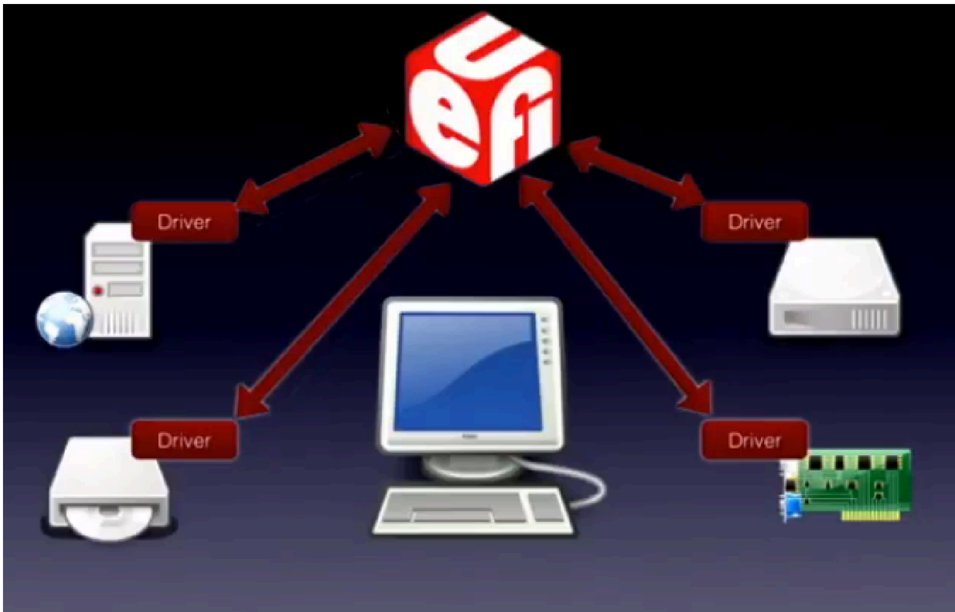
POSTED ON SEPTEMBER 28, 2018 BY ANISSE ASTIER

WELCOME TO THE KERNEL RECIPES LIVEBLOG ! THIS FOLLOWS THIS MORNING'S LIVEBLOG.

QEMU in UEFI — Alexander Graf

Running x86 servers is usually easy, and ARM servers found quite a challenge in the beginning when trying to run x86 code.

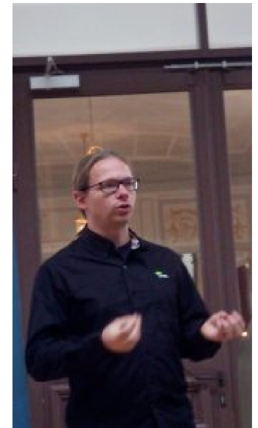
ARM and x86 differ with their register sets, their instruction sets, and aren't compatible at all. That's where QEMU comes in: it can emulate another CPU and allow you to run code compiled for another hardware. It also emulates all the hardware in a computer.



In UEFI land, you have drivers like in Linux, that give access to the hardware. They let you access network cards, disks, etc. Alexander says the UEFI driver interface isn't very well designed to separate virtual interfaces from hardware interfaces.

Alexander says that data structures in aarch64 (the 64 bits ARM architecture) are compatible to the ones in x86_64 servers: pointer size, padding and endianness are the same. But how does one call ARM UEFI function pointers from an x86 OS in QEMU ?

The trick is to use the NX (no execute) feature for the regions that might contain function pointers. Any jump to such a region would generate a trap, that can call the appropriate QEMU code that can translate the call ABI to call the proper function. And this works in both directions. This is emulation on function level boundary.



In QEMU, there's a code base called TCG (Tiny Code Generator), in LGPL, that can do on-the fly JIT instruction set translation, and support x86_64. Alexander says he didn't find any other projects that did the same.

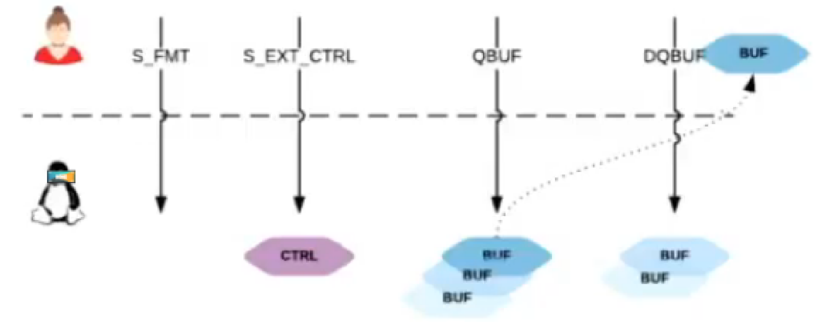
Alexander showed a demo of an emulated arm64 hardware, running UEFI, that loaded an x86 ethernet driver, to run an x86 OS loaded over the network.

He said that once the emulator hooks get merged in the next EDK, this type of emulation might start to appear in several ARM servers. In a response to the audience, he said that it will always better to have native code drivers, but having a backup solution is always useful.

Is Video4linux ready for all cutting-edge hardware ? — Ezequiel Garcia

No. Is the executive summary, according to Ezequiel.

V4L2 has relatively simple userspace API, he says. You configure the device, then request buffers.



Codecs are supported in the kernel, with the memory-to-memory APIs. Stateful codecs didn't have an interface specifications. The issue is that all drivers handle things slightly differently.

Stateless codecs relies on more work from the application to do extra parsing, and setup the hardware properly. In this case, the stream-based API isn't sufficient. For these drivers as well, the specification is a work in progress. The issue with the streaming API, is that there's no correlation between the controls and the payload you send and get. With the request API, this was fixed, and buffer and control metada can be synchronized. It's on the verge on being merged, and took four years of discussions, with many developers.

DMA Fences are a future area of evolution in V4L2, to provide explicit synchronization to buffers. It reduces the number of buffers to fill the pipeline, since they can go directly from V4L to DRM, without passing through userspace.

Asynchronous UVC should be reworked by Kieran Bingham to improve USB packate handling in multi-core SoCs.

Packet probes and eBPF filtering in Skydive — Nicolas Planel

Skydive is network topology and protocol analyzer with support for software-defined-networks. It provides a nice interface to visualize network topology, flows, routing tables, etc.

It helps with troubleshooting, flow monitoring, validation. Its dynamic UI uses d3.js with a physics feeling. You can request information on the graph with the Gremlin language. Gremlin is a graph engine language, with a Skydive implementation (not using tinkerpop).

In order to fetch information, Skydive relies on multiple topology probes: netlink, netns, ovs, docker, kubernetes, neutron, socketinfo, etc.

Skydive capture probes

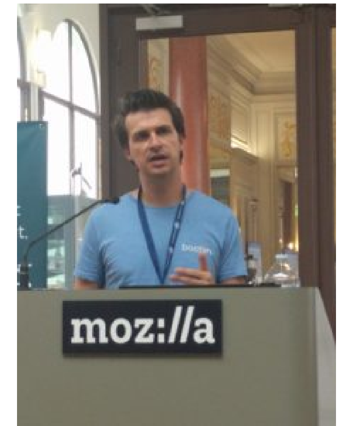
	Kernel	Overhead	Limitations
AF Packet	all	Huge	None
BPF (Syn/Fin/Rst/Ack/!Psh)	2.5+	Low	No packets metrics TCP only Start and end session
eBPF	3.18+	Low	No classification No fragmentation No tunneling (implementation)

The eBPF probes are fast, and run in O(1), Nicolas says. They have user space part (in Skydive), and a kernel space part. The userspace part matches the flows to the topology nodes. Skydive also matches the socket information to processes using kprobes and eBPF.

The roadmap include future hybrid capture, with the use of eBPF dissectors. Augmenting the information on local processes, like retransmission counters with eBPF and kprobes is also planned. Extracting the graph engine (Steffi) for outside use is also planned.

Overview of SD/eMMC, their high speed modes and Linux support — Grégory Clément

SD cards date back to 1999, then were extended with MMCs. They have 9 pins, and can work in SPI mode. The SD Bus protocol is composed of a command and a data stream. The transaction model is command/response, with all commands initiated by the host.



MMC evolved and diverged from the SD standard, with extensions from MMCA and JDEC. eMMC appeared in 2007, and is the embedded on-board version, designed with BGA chips in mind. eMMCs are now widely used.

Support in Linux for MMCs started in 2.6.9; SD card support was added in 2.6.14, and SDHCI arrived in 2.6.17. The first high speed mode for MMC and SD (with a clock up to 52MHz) was added in 2.6.20. SDIO was added in 2.6.24. The code is split between core and host parts, with the later being more hardware-specific.

New speed modes were added over time to be able to raise the base clock speed, via the CMD6 command. Starting with UHS-1, the new modes work at 1.8V instead of 3.3V, so voltage switching is required.

In MMC, the DDR mode, then HS-200 speed improvements were added in 2009 and 2011. Then the HS-400 in 2013, which works at 200MHz with a dual data rate and new enhanced data strobe line. The SD card also proposed new speed modes with UHS-II in SD 4.1, with a completely new set of signals.

Currently, eMMC speed mode is quite complete in Linux, with most development focusing on hardware-specific code. UHD-II and UHS-III aren't supported in Linux, but Grégory says no ARM SoC is supporting them either.

In June 2018, the SD Association announced a new improvement, called SD Express, based on PCIe and NVMe, which should be well supported in Linux.

That's it for Kernel Recipes 2018 ! Thanks for following the liveblog for this 7th edition and see you next year !