EPISODE 362

[INTRODUCTION]

[0:00:00.6] JM: The code in the Linux Kernel changes all the time. 11,000 lines are added, 5,800 lines are removed, and 2,000 lines are modified daily. Linux is an open-source operating system has been worked on for 25 years, and one reason that the project is able to move so fast is its governance and release structure.

Greg Kroah-Hartman is a fellow at the Linux Foundation where he takes part in many of the developments in the kernel. This episode was a dive into how open-source software gets built at scale and what is in store for the future. The Kubernetes project has drawn comparable attention to the size of Linux and the Kubernetes project is learning how to manage open-source from the Linux community.

If you're looking for old episodes of Software Engineering Daily but you don't know where to find the ones that are interesting to you, you can check out our new topic feeds which are in iTunes or wherever you find your podcasts. We've sorted all 500 of our old episodes into categories, like business, or blockchain, or cloud engineering, JavaScript, machine learning. We also have a greatest hits feed if you're just looking for the best episodes across all categories. Whatever the specific area of software that you're curious about, we have a feed for you. You can check the show notes for more details, and I hope you like today's episode.

[SPONSOR MESSAGE]

[0:00:00.6] JM: Spring is a season of growth and change. Have you been thinking you'd be happier at a new job? If you're dreaming about a new job and have been waiting for the right time to make a move, go to hire.com/sedaily today.

Hired makes finding work enjoyable. Hired uses an algorithmic job-matching tool in combination with a talent advocate who will walk you through the process of finding a better job. Maybe you want more flexible hours, or more money, or remote work. Maybe you work at Zillow, or Squarespace, or Postmates, or some of the other top technology companies that are

desperately looking for engineers on Hired. You and your skills are in high demand. You listen to a software engineering podcast in your spare time, so you're clearly passionate about technology.

Check out hired.com/sedaily to get a special offer for Software Engineering Daily listeners. A $600 signing bonus from Hired when you find that great job that gives you the respect and the salary that you deserve as a talented engineer. I love Hired because it puts you in charge.

Go to hired.com/sedaily, and thanks to Hired for being a continued long-running sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:05.1] JM: Greg Kroah-Hartman is a fellow at the Linux Foundation. Greg, welcome to Software Engineering Daily

[0:03:10.9] GKH: Great. Thanks for having me.

[0:03:12.1] JM: Describe your responsibilities in overseeing the Linux Kernel development.

[0:03:17.8] GKH: Overseeing — I have a role within the Linux Kernel community. I don't necessary oversee stuff. I'm a kernel developer. I'm a maintainer of different portions of the kernel and I also am responsible for releasing the stable Linux Kernel. Linus does a development kernel once a week and I do a stable kernel about a couple of times a week.

[0:03:38.1] JM: We'll get into what some of those terms, like stable kernel mean. Continuing with this overview of the Linux Kernel, Linux is 25 years old. How has the project changed over that period of time?

[0:03:52.2] GKH: Oh, wow! It's changed a lot. Basically, the biggest thing to look at how we changed is how we managed the project. A product used to be managed — We'd email Linus some and patches and we'd hope in a week that they would show up in his tree, because he'd release it hardball. We'd readjust and resend them in again.

Now, we use git. We actually create a git based on the need for the Linux Kernel. That came from how our development model worked, and we created this big giant ecosystem of developers. We have 4,000 developers last year that contribute to the kernel. Before, we used to not have that many at all. It's grown immensely. We've been able to scale and grow and we pretty much took over the world.

[0:04:32.9] JM: How is your role in the Linux project changed? How has it evolved overtime in tandem with the evolution of the scale and the tools that you've had available in the Linux project?

[0:04:47.1] GKH: Yeah, I started off as a Linux developer, then I contributed a few patches to the USB subsystem and then I wrote a driver for some hardware that I had and started maintaining that driver and then overtime started maintaining the subsystem for those drivers, which is the USB serial drivers, and then started maintaining USB drivers. I got a job doing this Linux Kernel work. Overtime, just started sucking up more and more driver work and contributed more and more things. Then eventually started the model doing stable kernel releases, and I can talk more about that, and I was one of the persons who started that and I've continued doing that since then over a decade now.

[0:05:26.9] JM: Yes, I do want to get into that eventually. The thing that's interesting about Linux — Well, one of the main interesting things, is that it's used in laptops, smartphones, a lot of other devices. I'm not sure if that was on the roadmap in the initial creation of Linux. Does the proliferation of these new device types affect the contents of the kernel?

[0:05:50.8] GKH: Totally. We have no roadmap. I want to make that clear. Everybody asks, "What's going to happen next?" We have no idea. We want to make whatever is going to happen today work well today and we'll worry about tomorrow of that.

Yes, we've made the kernel work for all devices, and it's a big deal. Everybody thinks their little problems area is special and unique, and I like pointing out the fact they're like power management, they've embedded devices. People came to us many years ago and said, "We need a good, better power management. We need to be able to control power better. We need

to be able to manage it better, and here's this neat patch and here's this change to the kernel and it's over in our little corner." People push back and said, "No, you need to make a generic for everything. All device need it." Maybe laptops might need it and some not.

It turns out, the biggest users of that are the mainframes, because they use the most power and there's like real money in the power that they'd use. You can shut the different parts of that down and we got it in the kernel and it touches everything, but it works for everybody. One change works for everybody, and that's what's made us evolved and worked really really well overtime because we forced developers to make it work for everybody.

[0:06:58.2] JM: That's true for something like power consumption, but what about something for self-driving? For example, being really good at image recognition? Is there something in the kernel that you might want to adjust to improve the image recognition, and that's something you wouldn't necessarily want in server aside Linux or maybe in android smartphones, but you want it in some domain-specific version of Linux.

[0:07:25.5] GKH: No, not really. I mean the kernel is written such that it handles all hardware out there in the world. That's how we really succeeded is because we work for all devices, but they're configured for all different types of devices. Your server doesn't configure the android code or your random audio device for that kernel. You can configure the kernel in many, many different ways. Audio or image recognition, you probably don't need — You only need one or two special drivers or the drivers for the drivers for the hardware that you're using and way you go. The kernel is actually for image recognition stuff. It's just a really dumb conduit. It just gets out of the way, and that's your GPU and everything else run fast.

[0:08:05.5] JM: Okay. You mentioned that there's no roadmap, and I find that interesting. There are some software projects that have really succeeded with no roadmap, like the Linux Kernel, but if you were talking about a company. If a company were to say, "We have no deterministic roadmap for the future. We're just latching on to whatever is kind of the local minima, or the local maxima of demand." The shareholders of that company might go crazy. Is there something about an open-source project or maybe Linux specifically where this modus operandi of not having a roadmap works?

[0:08:48.6] GKH: Yeah. Linux has contributed to many many people, what I say 4,000 individual people over 400 different companies last year, and every contributes in a very selfish manner. Everybody wants to get what their problem solved. As a maintainer of a subsystem of the kernel, we see it when people propose their solutions to these problems, be it new hardware, be it different domains that Linux needs to run in or larger memory or whatever. Companies have their own roadmaps to get their own hardware and their own solutions accomplished and then they post the code to us for our review, for our acceptance and critique based on that.

We kind of just go with the flow. We handle the new hardware that shows up. We handle the new code that shows up and we do that. We can't really plan for the future that way. That being said, we are told of what's — We know what new hardware is coming. Companies sit us down and talk about, "Hey, there's new stuff coming, like NVME, that memory bus stop we knew about years ago." We say, "Oh! That's great. Show us the hardware, or show us the drivers." It goes from there.

Open-sources need when you have lots and lots of different companies contributing, because they all contribute in a way that's specific for them but it works out for everybody.

[0:10:04.1] JM: I see, so you could trust in the long-term visions of those companies and find the intersection of highest cardinality among those things, and those will allow for something that resembles maybe a long-term vision that you don't even realize or a long-term vision that manifests in short-term decisions within the Linux Kernel.

[0:10:28.2] GKH: Sure. I talk to a lot of companies. I go around — part of my job with the Linux Foundation is to go and talk in to companies telling them how they can contribute to their kernel and how can they participate. The main reason companies want to participate is you or the company are using Linux in a very domain-specific way, be it like for a windmill, like all the big windmills use Linux, or a mega yacht, super mega yachts use Linux to their guidance control systems.

You've made changes to Linux and you want to get those changes back and accept it, because you have to trust — You trust yourself that you're driving Linux forward for you. If you don't

contribute back, you have to trust all the other companies that are contributing or driving it in a way that's good for you.

Now, ideally you do trust. You trust Intel and ARM and AMD and Qualcomm and all these C-chip manufacturers to make their hardware work well. What about the other core things, like the different scheduling ideas and people like that? If you're not contributing to that, then you're going to have to take their work for it and take their code for it, which is fine, and a lot of companies do trust that, but I try and say, "Wouldn't it be nice for you to control your own destiny and participate?"

[0:11:32.8] JM: You are an advocate of a stable API between kernel and user space. I think I cribbed this directly from Wikipedia. There's a lot of people listening and I would actually put myself in this bucket who really don't know a lot about Linux. Like we've just worked with Linux as people who are building web apps, and this is like the thing that we're sometimes deploying our web app on and beyond — We know, okay, you can use Sed and Awk and some other stuff on that command line, but we don't really know much about what goes on underneath.

Just so we can have a little bit of a common language, there is this kernel and this user space. What are the responsibilities of these two areas of Linux?

[0:12:20.6] GKH: The kernel is there to do — Basically, all a kernel does is manage resources and it makes your hardware look like something in a very consistent way. It makes all mice look like all other mice. A kernel makes the hardware look in a common way so you can write your applications and it will run on any Linux machine that's out there. You don't really care what the underlying hardware is. You don't care what the disc is. You don't care how much memory you have or what kind of keyboard you have. They all just work. That's one job of a kernel is to manage all the hardware and to manage the resources, to manage your files, to manage the file system that you're backend — Your storage is on, be it a local disc, be it a remote disc, be it a cluster, be it whatever else. The kernel is there to abstract and give you a common API in which to work with these resources, be it files, or memory, or a mouse.

Yes, we need a very stable API there. We need an API that you'd write a program and it's going to work for forever. That's our main goal is I have programs that are written in the 90s, that are

compiled in the 90s. They still work just fine on Linux. That's our goal. We want to keep that API stable.

In fact, that's the only time when Linus gets mad and curses at people is when that API is broken on purpose. We accidentally break it. Great. We'll fix it. We accidentally break things all the time, but if we break it on purpose, then we get mad and then the cursing happens, but that's the only time.

[0:13:47.8] JM: When Wikipedia refers to you as an advocate of a stable API between kernel and user space, this sounds like something that is non-controversial. Is it the fact that you're an advocate in response to perhaps when somebody submits a push that is not — It's not like they're — Or a pull request that is not necessarily — It's not like they intended to have a fuzzy API or an inconsistent API between kernel and the user space, but because they did a rush job or something. You have to reject the pull request and say, "This is just not cleanly partitioned." What does it mean that you're an advocate of this thing that it sounds like everybody would take for granted?

[0:14:29.3] GKH: I don't know. I haven't read my Wikipedia page in a long time. In some areas people — We make mistakes. It's really hard writing — In the early days, we all were choosing Unix and we all know what we needed to do. That day has been passed like about 10 years ago and we're in the forefront now creating new user space APIs to solve problems is hard and we get it wrong usually. Getting it wrong means we still got to keep around the old one and make it modified in a way that's not going to break somebody.

Michael Kerrisk who maintains our man pages, which is a documentation of this user to kernel API, has a really really good talk out there on how to make a good API and how to make a good API in a way that's going to be future-proof and that if we mess something up, and how we can recover and fit it and go forward. He's given that talk at a number of Linux conferences over the years. He's also written a really really good book of the Linux programmers API. I don't remember what the name of the book is, but it documents this whole API between the user and kernel space.

I wrote a document a long time ago in the kernel talking about the internal API that the kernel has. A lot of people advocate the internal API between the kernel and drivers should be stable so that they can write a driver once and not work for forever. Other operating systems, they claim have stable APIs for drivers, that way you can have your driver outside the main tree and it will still work.

I'm a strong advocate that that is not the way to go. In Linux, we refactor our internal API very very hard. We do it with total abandon. If we see something that needs to be changed, we'll fix it up everywhere and keep on moving. Our internal API changes all the time. The main goal of all drivers in the kernel and all code needs to be merged into the main kernel, because if I change an API in the kernel, I'll fix it everywhere. I'll fix it in your driver, but also see commonalities.

If we have free drivers for something that almost looks like the same hardware, hey, we can merge all these tree together, save code, save space, and move on, and we can reiterate and refactor. Because of that, Linux drivers are about one-third the size of other operating system drivers, because we can do that, and one-third the size is great, less code, that can be less memory and less bugs. I'm a strong advocate for stable API outside the kernel, but a very broken one inside the kernel.

[SPONSOR MESSAGE]

[0:16:58.6] JM: Angular, React, View, Knockout, the forecast calls for a flurry of frameworks making it hard to decide which to use, or maybe you already have a preferred JavaScript framework, but you want to try out a new one. Wijmo and GrapeCity bring you the How to Choose the Best JavaScript Framework For Your Team e-book.

In this free e-book, you'll learn about JavaScript frameworks. You'll learn about software design patterns and you'll learn about the advantage of using frameworks with UI libraries, like Wijmo. You'll also learn about the basic history and the purposes of JavaScript's top frameworks. You'll also learn how to integrate a WIjmo UI control in pure JavaScript and in some of the top JavaScript frameworks that we've already were talked about, like Angular, React, View and Knockout.

Wijmo's spec method allows you to determine which framework is best suited to your project based on your priorities. Whatever those priorities and your selections are, you can learn how to migrate to a new framework in this e-book. Best of all, this e-book is free. You can download your copy today to help choose a framework for your work at softwareengineeringdaily.com/grapecity.

Thanks to GrapeCity for being a new sponsor Software Engineering Daily, and you can check out that e-book at softwareengineeringdaily.com/grapecity.

[INTERVIEW]

[0:18:41.1] JM: I just put a note to send an email to Michael Kerrisk, because that sounds like quite an interesting topic, his discussions around APIs. We've had a lot of shows about microservices and distributed systems and how companies scale that because it's a hot topic. You're talking about keeping a contract between two maybe divergent systems clean but aggressively updating an internal API. There's a lot of discussion around these distributed systems whether you should version APIs and when and if you should version it. Do you ever do versioning where you maintain an old API and then you have certain things that are making new requests?

[0:19:30.7] GKH: Oh, totally. We have like [inaudible 0:19:32.8] that have a two at the end of them, because we've learned over the ages — And even POSIX. I think POSIX has a few of those as well. They defined that APIs, "Oh, we need to make a new one, and here's the second one," or "Here's the open with additional parameters and it open at functions that are regular open." Yes, we do that all the time. We add another one and moved on.

Then internally, you can usually rewrite the old call as subset of a new call and it works just fine. Yes, we have to maintain that. We provide that guarantee because they want people to be able to update their kernel without having anything break. We don't want you ever to have to rewrite your code because of something we did, because that's just mean. I don't want you to have to do that. That's something that a lot of software projects fail at and that's hard. It's hard to do. I agree. We're lucky and that our API is very tiny, relatively speaking. We don't have to have as

big of a problem there, but it's very important if you want users that are going to continue to use your product.

We also guarantee that we're not going to break any breaker system, so we want no regressions. We will always fix a regression over a new feature. We'll rip out a new feature that broke something and put it back the way it was at a drop of a hat because we do not ever want you to be able to — If you upgrade your kernel for something to break. We never want that to happen.

People tested that. I think Facebook tested that a long time ago for the stable kernel releases over like 90 releases that I've made with stable kernels. We never broke anything. We made that guarantee about over a decade ago when we've been doing really really well there.

[0:21:09.0] JM: I want to discuss the governance and the contribution process of Linux. You've defined this term stable kernel a couple of times, or you've mentioned it. Could you define it?

[0:21:23.4] GKH: Sure. I talked about how our development model works because it fits into that.

[0:21:27.1] JM: Sure. That will be great.

[0:21:28.2] GKH: Okay. You start off from the developer, or somebody wants to make a change, be it you see a spelling fix or you want to add a new driver per se, you got a new type of a mouse. You make a change to the kernel and then you email it in, and those developers, we have about 4,000 developers last year contributed to the kernel.

You email your path on a mailing list, not using git. You can use git internally, but we don't take pull request, so we just want email. We use email because it's scales. It scales much much better. We're averaging our acceptance rate for patches is about eight patches an hour and we only really accept about one-third of the patches that are sent to us. Two-thirds larger, number of the patches that's sent to us.

Email. You send email to the patch where you change to the owner of that driver or that subsystem, and all drivers and subsystems in a kernel have an owner. It's somebody who wrote it first or, say, USB, I maintain the USB subsystem, but the USB serial subsystem which is a subset underneath me is maintained by somebody else.

You send it off to there, there's a public mailing list for every different part of the kernel. At small mailing list, there's a huge giant mailing list, Linux Kernel mailing list that we all kind of filter. Nobody reads it. That's our little dirty secret. It gets about 400 emails a day. Those people send a patch off their email, it gets reviewed, it gets accepted or rejected. You revise it, resend it, and then eventually the maintainer of that subsystem accepts it and then — Maintainers of subsystems, we have about 150, maybe 200 of those. No, sorry. 700. 700 maintainers.

Then they have a kernel tree and then they punch it up to a large subsystem tree and that we have about 150 of these subsystem trees. These are all public on git.kernel.org and all the patches go into there and then they emerged once a day altogether by somebody in Australia, Steven Rosswell merges them all together and produces what we call Linux Next, and that is where we see all the problems that are going to happen. This is our development tree, and we see where the mergers happen, the problems.

For example, if I make a change in a network driver, in my USB subsystem, the network maintainer is making an API change, will see that error there, will see that problem. That's how we do it as far as what we do every day. These trees get built. They get booted on virtual machines and see how if anything shakes out.

During the merge window, and I can talk more about in a minute. Linus has a merge window when he does a release. For two weeks, all the maintainers of this subsystems send all these patches, all the patches that we have ready for him to him, and then Linus does a new release. We call it release candidate one, RC1. Then for every week after that, Linus does a release candidate, release candidate two, three, four, five, six. Once a week, he's doing them on Sundays. Every release he does from after release kind of one is bug fixes only or regressions or revert something. We'll rip something out. Bug fix is only new release candidate for up to about release candidate seven or eight. Looks good and solid and stable. Linus will do a new

release and then the whole thing starts over again. The maintainers throw all the new stuff at them and a way we go.

We do a new kernel release about once every two and a half to three months, and we've been doing that for about 10 years. We do timed releases, so we do a timed release this way. Time releases are great and that as a maintainer, I don't have to feel pressure to accept a patch from somebody that's not in a good enough state because if I can't accept at this release, I know three months from now it can get it in. It's good, because if I had a two-year window, or even a one-year release window, I'd feel a lot of pressure because that's a long time to wait between releases. That's how we do our development releases. Linus has a new release and a way we go.

Now, about a decade ago, we realized what these releases from Linus, during those two and a half months, bugs happen. We got something wrong, security bugs happen. Something else happened. What do we do? I'm the maintainer of these stable kernels, so I fork off Linus' tree and then I take patches that are in there for stable kernels. I do a stable kernel release about once a week. I have some cool drawings about how these all works. I can show you in a presentation as well.

The rules for the stable kernel are the patch has to be in Linus' tree, and we make that rule that it has been in Linus' tree so we never diverge. We never want to patch to go into a stable kernel that isn't in Linus' tree, because when you update to the next version of Linus' tree, you also want your fix to be in there as well. I do stable releases. Then after Linus has a new release, I will usually drop that old stable kernel and then go on to the new one. That's how we have stable kernels.

People like these table kernels and they build products on them. If you look at your distros, your distros are all based on these stable kernels. Phones and android devices are being built off the long-term kernels, and long-term kernels are something that we've picked up over the years. I'll maintain one of these stable kernels. I'll pick one of them a year and I'll maintain it for two years. We back-port all the fixes to the older kernels and then that way older devices, devices that are a year or two years old will constantly have stable or bug fixes and security updates as time goes on. That's really a good thing.

[0:26:41.5] JM: Could you zoom in on relationship between release candidates and stable kernel? Maybe define release candidate for the audience.

[0:26:51.5] GKH: Release candidate is — Linus' tree, we throw a whole bunch of stuff at him for two weeks, about 9,000, 10,000 patches, and so he does, "It looks good to me. Let's get a release candidate out there." These candidates are what the next major kernel release is going to be. Then it's the accumulation of about two to three months worth of work for people. 10,000 changes is about usually per kernel release. That's a release canvas. It has new features, new drivers, new subsystems. Lots of new stuff.

Stable kernels are bug fixes only for the last major release. The rules for stable kernel is it has to be a fix that's obviously correct. Usually less than a hundred lines long. A fix is something that people report, or it can add a new device I.D. Say, you got a new type of mouse, it just adds a new item to a table somewhere. We'll take those as well. Stable kernels just get bug fixes only, and that's it. No new features. No new additions like that.

[0:27:49.0] JM: Throughout this release process, what tests do you have in place? Do you have automated tests, manual tests? What's going on in the submission process and testing?

[0:28:02.1] GKH: Tough. That thing is interesting. A lot of people are finally waking up to that one. Many many years ago, we used to have a lot of test going. A lot of companies are running tests and they came up with the Linux test project, which was written a cover like all sys calls, some driver subsystems had some tests. There is some stress test for networking and a file I/O.

Overtime, everybody thought everybody else was doing the testing, so they stopped testing. About four or five years ago, we realized nobody was testing, and that wasn't good. Intel stepped up and they created something — An engineer there created something that we call the zero-day test bot. I think it's just it's just this giant giant server farm of CPUs that are idle.

What he does is he goes through all our developer trees, all the git trees, and he boots them all, build them all, boots them all, and started running tests on them all. Booting a kernel is a non-trivial task. If a kernel boots, they're usually pretty good, but we boot and build tons of different

configurations, architectures, and then he runs tests. I think there's a whole bunch of performance test on there to run for longer periods of time.

This bot that's out there is amazingly fast. I can push a git tree out and then an hour later I'll get something back saying, "Hey, out of these hundred patches, you just pushed patch number 65." I have this build warning over here in patch number 72. Broke the build in this way. Go fix it. It's amazingly good.

Those are build tests. We also have a lot and lot of static analysis test, and static analysis is you run through the codes just to check the code. You don't actually run it. These will look like a C compiler. We have something called Sparse that Linus created which is a C front-end. There's something called [inaudible 0:29:40.4] here in Paris, which is one reason I'm here in Paris. You can write rules for this tool that analysis the source code for common patterns that we know are wrong.

She's written tests for the kernel that have fixed more security bugs than anybody else because of this. Once we write a test, we add it to the test suite and then that prevents any new code that comes in with that same kind of pattern from ever being able to be introduced again, because we catch it, and we have hundreds of those tests.

We also have tests in this test subdirectory of the kernel. We have a maintainer of these tests now, and they tests specific functionalities. If you add a new sys call to the kernel, we can exercise it. Make sure it works properly. Between all those tests, all the developer trees are tested, Linus' tree is tested and Linux Next will merge all these together are all tested every day. It's really good. We test a lot of different things.

[0:30:31.6] JM: Do you feel comfortable with the automated test coverage or do you feel like there is also a necessity to do a certain degree of dog-fooding and perhaps manual testing?

[0:30:46.1] GKH: Oh, we have to do manual testing. I'll give an example. It turns out that we have a test we called Linux Test Project, LTP. We finally ran it on some ARM 64 machines and old kernel and found a bug there introduced over a year and a half ago. It turned out everybody

who's running it on emulators and never on real hardware and there was a timer bug. It was a very nuanced hardware resolution timer bug, but they would only show up on real hardware.

Yes, you have to debug, or you have to dog food on real hardware because there are — There's also flaky hardware. You can only do so much with emulation. You can do a lot with emulation. The graphics guys have some amazing emulation that they do for graphics test. You really need a real hardware. There are people out there.

There's a group called Kernel CI from Bay Libre is the company, and Linaro also contributes, and they test and they have a couple of hundred different ARM machines that build and boot all the kernels all the time, my stable kernels, Linus' kernels and a whole bunch of other things. The ARM developers have a whole bunch of test systems that they've build into the ARM. You do need real hardware as well. I test all my kernels and stuff on my laptop. Again, booting a machine, a laptop, is a non-trivial task.

[SPONSOR MESSAGE]

[0:32:03.8] JM: The Women in Tech Show is a podcast featuring technical interviews with prominent women in technology. The most recent episode features Karen Walker, former VP of Operations at Compaq. She talks about working on the largest supplier of PCs in the 90s. With the Compaq Portable, they took on IBM and became one of the fastest growing companies in American History.

Other great recent episodes have covered subjects like robotics, distributed systems, and ethical questions of artificial intelligence. Check out The Women in Tech Show on iTunes or wherever you get your podcasts.

[INTERVIEW CONTINUED]

[0:32:51.8] JM: At companies, the wildly accepted strategy of breaking up the company into teams of different projects, it's not universally accepted, but there is a lot of companies that do this two pizza team thing where it's like maybe 10 people max, or 12 people max on a team. It seems like that might be less important on a completely decentralized project. Could you just

talk some about how teams or specific modules in the Linux Kernel are managed by different people? I'm interested in the dynamics of the different sub-teams involved in the Linux Kernel.

[0:33:37.8] GKH: Sure. The Linux Kernel, you can think of it is a pyramid. Say, there's 10 USB developers and then they send patches to me. I send patches above that and we go and approve that, like the networking system, there's wireless developers, they send it to the wireless subsystem maintainer, he sends it off to the main thing or networking and then send it off up to Linus. It's a bit of a pyramid there, but we're all people. We're people, we have human interactions and we try and we've learned over the years that we need to get together and talk and meet each other because all we ever used to do is see each other through email. About 12 years ago Ted Ts'o finally realized that and created the Kernel Summit where we all got together and meet each other for the first time.

It's a hierarchy of people accepting changes from other people, passing them on up this stack. I take patches from people I know over the years and I trust and I don't necessarily trust they got it right. I trust that they'll be around to fix it if they got it wrong. That's the important thing. You have to trust somebody who's going to be there to fix the problem, because when I accept your changes from somebody that submits it to me, I'm not responsible for it, because my name is on it. I signed off on it. It goes on up the chain that way. Hopefully you're going to stick around, because I have enough code in my own to maintain.

It's a personal dynamic. It's meeting people and realizing that they're going to stick around and do that, which is hard for some subsystems. Some subsystems use a kernel, it's very hard to get changes into it. People don't trust you. They don't know who you are. Networking is a huge example, scheduler is a huge example. Networking is a huge — It's proven itself. Many years ago, a new feature finally landed in the networking subsystem, a big, hairy, nasty feature, and the day after it hit the tree and was finally merged, the email address behind that disappeared and it took the developers six months to unwind the mess and fix it all up probably.

It is very very hard to get a networking core change in for somebody who's brand-new. It's not that we don't necessarily trust that you got it right, but if you got it wrong, are you going to stick around and fix it? You have to accrue trust and you have to start off small, start off by reviewing other people's patches. Start off by integrating yourself, go to the conferences. We all travel. We

all meet each other. Say, “Hey, look. I’m a real person behind this. I’m looking to do this. Talk to us, and go from there.” Yes, it is managing. I trust some people, people above me — Linus trusts some people. Again, yeah, 10 people is about how much a normal team you can work with.

That being said, there are some of us maintainers that do accept patches from a huge number of different people. A networking maintainer ends up accepting patches from large numbers. I end up accepting patches from large numbers to some of the subsystems I maintain where there are a lot of just drive-by fix up a tiny thing and go away, which is fine. Those changes are great. We love those changes, because we never know where our new developers going to start from. A nice, easy, simple spelling fix it’s a great place to start, and they can like that and keep them going on to something else.

[0:36:40.5] JM: I believe there are some interesting lessons in company management that can be derived from the Linux Kernel management. Again, not lots of direct conclusions, but certain, perhaps inspirational things. Do you have any ideas around what is the — You talk about this pyramid where there’s somebody at the top and then there’s different layers of people who are — Changes are bubbling up and these different people at each layer of the pyramid are aggregating those changes and checking them against other people. Do you have any beliefs about the branching factor of that pyramid or how deep you can go before it become just too many layers of management between the top and the bottom?

[0:37:26.3] GKH: Yeah. It’s not as deep as it sounds. I actually graphed this one year. I posted it was a giant, like two meter tall by 10 meter, 20 meter long graph. It was a huge mess. We talk about it as a nice neat pyramid, but the routing of way patches go through the kernel is crazy. It’s a huge giant network graph, which is good, because sometimes being a maintainer of a subsystem, it’s not the ability for me to say no to something. The networking maintainer can make changes to the USB subsystem if he things that’s what needs to be done.

If he’s nice, he’ll ask me, “Do you think these are okay?” I’ll say yes or no and it will go from there, but it isn’t an absolute control. Nobody has absolute control over their subsystem, and that’s a good thing, because people go on vacation, people get sick. People pass away. We’ve

had maintainers that die. Life happens. You can be routed around, which is good for the stability of the project.

That necessarily isn't the same thing as a company. You can't usually have that. We don't really have managers per se. We have people that are maintainers of subsystems and we impart taste on how we know things work based on our experience. That's also another thing Linux has and no other operating has. We have people that have been working on these things for much much longer than anybody else.

At a traditional company, if you stay five years in the same group, something is wrong. You have to move on and learn other parts of the company. I've been doing USB since the 1990s, and networking maintainer has been doing networking since the early 90s. If you look at the depth of knowledge of what we have in Linux Kernel, it's crazy how long people have been working on these domains and knowing it, because we even move to different companies and we take the subsystem or we take the maintainership with us, because it's done as individuals, not as a company. That's something that companies need to learn when they join the Linux Kernel.

One company I worked at, somebody was assigned to one subsystem and he ended up becoming the maintainer of it, which is great. Then he was supposed to go — His company wanted him to go off and do something else. We're like, "No. No. No. Wait. He's still the maintainer of this subsystem. He still has to do this." Then he did that and they had the car at the time and then he changed companies later on and then he took that maintainership with him and he's still doing that same type of stuff because he enjoys it, but the knowledge is there and the depth of knowledge is there.

We respect each other, but we can touch, we can modify each other's code. We can be routed around and things work that way. It isn't a traditional development model. That being said, I think hardware business review has written a number of papers. I used to joke that a lot of people should be able to get master's degrees based on look at our development model. It has. I think there's been a number of different papers have been written on how our development works over the years, which is great to see.

[0:40:11.4] JM: I read an article about your work with the Kubernetes team. How was the Kubernetes project similar to Linux?

[0:40:21.6] GKH: They're similar that they're really successful right now. Also, they had the wall on a number of developers wanting to contribute really really fast. GitHub is a great place to work maybe if you have 10 people on your project, but Kubernetes has so many people under a project. GitHub does not scale well. Docker hit this wall a long time ago, and I worked with a Docker community to try and help them figure out to change their development model.

Kubernetes had it really really fast, so they had a very very short-term to get their ecosystem change based on — We've had the kernel evolved this way over 25 years. They hit the ground running, so I feel sorry for them in that sense. That being said, they're doing a really really good job on trying to handle this stuff. They still have growing pains, I still talk to them a lot. I help out advising them when they ask any questions. It's tough. It's tough to scale a project really really fast. We've had experience doing this and it slowly evolved overtime. They were having to do it very very quickly.

[0:41:24.7] JM: You've said this a couple of times about how a repository management tool like GitHub might not scale if you have a project that's big enough. What are the specific pain points of anything that is not email basically?

[0:41:42.4] GKH: I actually gave a good talk at the Kernel Recipes Conference last year about how all these tools suck a high speed development besides email.

[0:41:52.5] JM: I assume you're including Gitter and Slack and all these different — Maybe IRC.

[0:41:58.3] GKH: Slack is IRC. We all use IRC, there's kernel IRC channels. There's a kernel newbies channel, which is great if you're learning kernel development. There's a kernel newbies IRC channel. There's like 400 people in it. Nobody's talking. Just ask your question and it will be answered. No, we use that Slack, it's fine. I'm talking about things like Gerrit. Gerrit is horrible for submitting patches and review. I know people use it. I've been dealing with android

developers. I'm getting more and more used to it, but it's still pretty bad. It doesn't scale very well.

For GitHub, it's hard to do patch review. Patch reviews in siloes of this individual patch. It's hard to see if you maintain an area of the kernel. You can't just watch everything go by and contribute where you want to in an easy way. You have to click around, drill down and do it. That being said, GitHub has worked a lot. They've worked a lot to try and make it easier. You can now use email along these reviews on GitHub, but it's hard. It's a hard thing to do. Not everybody can just use emails because there are managers out there who want to see what the outstanding issues are and where things are — What state certain patches are in and so on.

There are some really good tools that people have written. There's something called Patch Work that works on top of a mailing list and a number of kernel subsystems use this, like networking. When you email your patch, you can then go and look on a webpage and see what the state of the patch is. You can see has it been reviewed? Has it been accepted? Yes or no? Where it is in the response? Do I need to do something else with it?

There are tools on top of email that work really well also. People are used to thinking of email and some client like Gmail, which is a horrible horrible client as far as dealing with large numbers of email. You need a better email client. There's a lot of better ones out there. We've been using this for a long long time. For example, I get about a thousand emails a day for a non-mailing list stuff that I need to do something with. I can handle that trivially in my email client Gmail would just choke hard. I could never do that.

[0:43:52.1] JM: You talked earlier about how other changes to the world impact the Linux Kernel cause changes to be made in the Linux Kernel. Is the rise of Kubernetes come back to impact the Linux Kernel?

[0:44:09.4] GKH: Not that I know of. Kubernetes is just like basically an application on top of containers that manage containers across the network. That's a great useful tool. It's very helpful. I don't pay attention too much in networking subsystem, but I know the networking name space is and there's been other name space. Kernel patches over the years that have helped

these people out, have helped containers out, has helped Docker out, and by virtue of that has helped Kubernetes out.

That being said, we do run containers very well in those. Thanks to low-level kernel changes that were made years ago. I'm sure the bug fix is coming every once in a while as well.

[0:44:49.2] JM: I'm someone who has seen containers get a lot of headlines and be used as this distributed systems management unit that gives me some better usage of my infrastructure. As I understand, containers, they've been around for a longtime. It's nothing new. What's new perhaps is the usability or the applicability to managing a distributed system. Can you talk about how containers have played a role within Linux? Do I have containers running on my desktop Ubuntu? Are they doing something just in a single system?

[0:45:32.7] GKH: No. They're not. The single systems are not. Some people are working on to change that, because there is a nice partitioning and a tiny bit of a security barrier you can put between applications that way. I think android does something like containers. I'm not quite sure how their user space works for applications, but their applications are very siloed in a container-like way. I think when you run Android on top of Chrome OS, those are using containers as well. A regular Ubuntu desktop, open [inaudible 0:45:59.8] or whatever, no. You're not using containers.

The weird thing about containers is a lot of the work that's done with those to handle this container can't go over so many percentage of my CPU happened in the kernel years and years ago. It's something called C-Groups, which is container groups. That work came from IBM and some other places where large systems were trying to be divided up into smaller ones and they wanted to be able to keep — They wanted to be able to partition stuff away.

The S3-90, big giant name frames are on Linux drove a lot of that work. I always joke, if you want to see where Linux is going, look at what the S3-90 are doing, because they seem to solve our problems about five years in advance of everybody else. They did the container work first. They did Hotplug first. They did 10,000, 20,000 scuzzy devices first over the years. These things came out of the main frame world. Now, they work for everybody, which is great.

[0:46:50.7] JM: Piecing together a few things that you've just said, and you call tell me if this is incorrect, but are you saying that the idea of a C-Group — When I hear people talk about containers, okay, container is a name space in a C-Group, and then what you said is C-Groups are this way of — It's like an Outlook. It's like allocation of memory or resources that you're giving to an application or something that's running and you said the idea of a container, like a Docker container, is really more about the networking among different systems, and that's why maybe you don't necessarily need this abstraction of a container on a single machine because the networking architecture going between different machines is really the big deal with different container. Am I getting that correct?

[0:47:42.7] GKH: Kind of. What Docker did really well — And everybody says Docker is just a really simple thing. What they did really well was that they pulled together the fact that you can have different network name spaces. Say, on a machine, you can create a network name space up. This network is only talking to these processes and it's going to look separate from the other processes. They can't see the same networking connection as well as a file system name space. You can mail the file system in a specific location on the disc or in the large file system and you can't see outside of it, so you can't break outside of that area. Combined with things like process isolation and the C-group stuff were saying, "This process, I'm only going to be able to allow it to never go over 50% of my CPU usage right now. I'm never going to allow it more than two gigabyte of RAM so it can't take anything away from anything else."

Docker bundled all that up together in a cool tool that said, "Here's your file system. Here's your network. You're creating these name spaces and here's the resources you're going to provide for this, and away you go," and it runs one process inside that "container" that's network isolated, it's file system isolated and it's memory in process, another resources isolated. Docker — And then by virtue of Rocket and these other container engines, bundle that up in a nice way. Kubernetes on top of that just makes it so you can take your container which is just one process and move it around the network in a resource way because you want to have 20 mail servers running it on things like that.

The kernel provide these little tiny building blocks, Docker and Rocket, then combine them to another building block that you run your process inside off that Kubernetes then manages on top of that and so on and so on.

[0:49:24.7] JM: One of the themes of computer science, and it's a been theme in Software Engineering Daily is the idea of schedules and scheduling, and this is a problem area that's just evident at all layers of the stack. What have you learned about schedulers from the Linux Kernel, and has there been any application from that knowledge-base that you've gotten from working on schedulers in the Linux Kernel in your conversations about Kubernetes?

[0:49:55.1] GKH: When I think of a scheduler, I think of a process scheduler. On the kernel, Linux is a multi-process machine. The goal of the scheduler on the Linux Kernel is to run as many processes that are waiting to be run as possible. Other process is waiting for some I/O to happen or some networking — Yeah, basically I/O file or disc or network to happen. I don't want to run, because I'll just sit and busy wait. So a scheduler's job is to run everything as good as possible.

The Linux Kernels had a number of different schedulers over the years. If you're looking at your device, your phone, they usually wrote a different scheduler for that because phones, devices, want to — Takes power management into the scheduling equation. They want to be able to make sure that the power usage of a certain process, that this process is going to take a lot of power, we need to run it at a certain way. They modified a scheduler based on that, and then there's changes upstream that are going in to slowly make that happen. You can have processes that have different priorities, be it like a real-time priority where you have to be accessed every X-number of time or I want to run as fast as possible, then get me out of the way, or just run whenever you can, and there's a free time in the system. That's what I think of schedulers.

There's also networking schedulers to handle networking queues; and disc schedulers, they handle all your disc queue, because a disc is really just a whole bunch of packets going down that are being read and written to a device. You need to be able to schedule those. You can 10 times call some of the larger requests or split it up into smaller ones and go on from there. The kernel has different networking or different networking schedulers. They have different block schedulers. They don't have networking. I don't networking that well.

When Kubernetes talks about scheduling, that's a totally different type of thing. In a way, it managing resources of processes. You want to be able to make sure that you have 20 mail servers running somewhere. If one disappears, where do I schedule the next one to run? It's a same idea of resource management, but it's just running processes in different places.

[0:51:51.4] JM: You've mentioned this dynamic that corporations have with Linux where the corporations have their long-term goals and those long-term goals get aggregated and they kind of manifest in short-term decisions in the Linux Kernel and asymptotes towards a long-term vision for those different companies. Give a sense for how that compares to the corporate interaction with Kubernetes. Because, certainly, look at Kubernetes, and there is a frenzy of opportunity around — Companies that are being built around Kubernetes, on top of Kubernetes, making enterprise distributions of Kubernetes. What's the interaction that you see between corporations and Kubernetes? Is it the same as what it was with Linux?

[0:52:40.6] GKH: I hope it's the same, because — Companies contribute to the kernel and they participate in that because the kernel is not something that you compete with. It's a common shared resource, just like Kubernetes is. These companies are contributing to Kubernetes to make things work better for their customers and they're not trying to make something different from anybody else per se. They're competing on different areas, like Linux, Red Hat, and [inaudible 0:53:04.6] grew up over the years by competing with service and how they managed kernels for their customers and how they provided service for other people. IBM and Intel contribute to the kernel because they want to sell hardware. That was their main model.

ARM contributed to it. Other people contributed in different ways because they want to sell hardware services, but you're not differentiating yourself on the base product. Nobody differentiates themselves in the kernel because, again, it's a low-level common good. Same thing with Kubernetes, everybody sees that there's a lot of work to be done and they're contributing. A lot of companies are contributing to Kubernetes, so it's great. Companies are getting started based on Kubernetes because that's a resource that companies want or that other companies want to use. You can provide Kubernetes hosting for people, you can provide consulting for doing it in-house, wonderful things like that. Again, it's a common shared resource. They're competing with it, they're just participating and making it better for everybody. You can't compete on the commons, right? It's a commodity.

[0:54:02.7] JM: Okay. As we begin to draw to a close. I've really enjoyed this conversation. If you woke up from a dream right now and you realized that this whole Linux thing had all been imagined and you're back in 1991. Nobody's come out with a wildly adopted open-source operating system. If you were starting this project from scratch, the Linux Kernel, what would you do differently?

[0:54:27.8] GKH: Oh, man! We are lucky that we succeeded. I always say that. Yeah, Linus picked the right license, it was at the right place at the right time. There is a deal in the market of innovation in certain areas. We had white box machines come up and Linux came from below inside companies and grew up from there. I'd say we do it again.

We always joked that we were going to — Our goal is total world domination. It really wasn't a joke. We did it. I think we did a pretty good job. We got lucky. We got a lot of really really talented people that contribute to the kernel over the years and are still contributing. We're very lucky in that. I'd say do the same thing.

[0:55:09.7] JM: All right, Greg. Greg, thanks for coming on the show, and thanks to Dan Conn for introducing us. That's why I got confused there, but thank you so much. I really enjoyed this conversation.

[0:55:21.8] GKH: Great! Thanks! I had a fun time.

[END OF EPISODE]

[0:55:30.8] JM: You have a full-time engineering job. You work on back-end systems of front-end web development, but the device that you interact with the most is your smartphone and you want to know how to program it. You could wade through online resources and create your own curriculum from the tutorials and the code snippets that you find online, but there is a more efficient option than teaching yourself.

If you want to learn mobile development from great instructors for free, check out CodePath. CodePath is an 8-week iOS and android development class for professional engineers who are

looking to build a new skill. CodePath has free evening classes for dedicated experienced engineers and designers. I could personally vouch for the effectiveness of the CodePath program because I just hired someone full-time from CodePath to work on my company Adforprize. He was a talented engineer before he joined CodePath, but the free classes that CodePath offered him allowed him to develop a new skill, which was mobile development.

With that in mind, if you're looking for talented mobile developers for your company, CodePath is also something you should check out. Whether you're an engineer who's looking to retrain as a mobile developer or if you're looking to hire mobile engineers, go to codepath.com to learn more. You can also listen to my interview with Nathan Esquenazi of CodePath to learn more, and thanks to the team at CodePath for sponsoring Software Engineering Daily and for providing a platform that is useful to the software community.

[END]

