Sign Git Commits with GPG

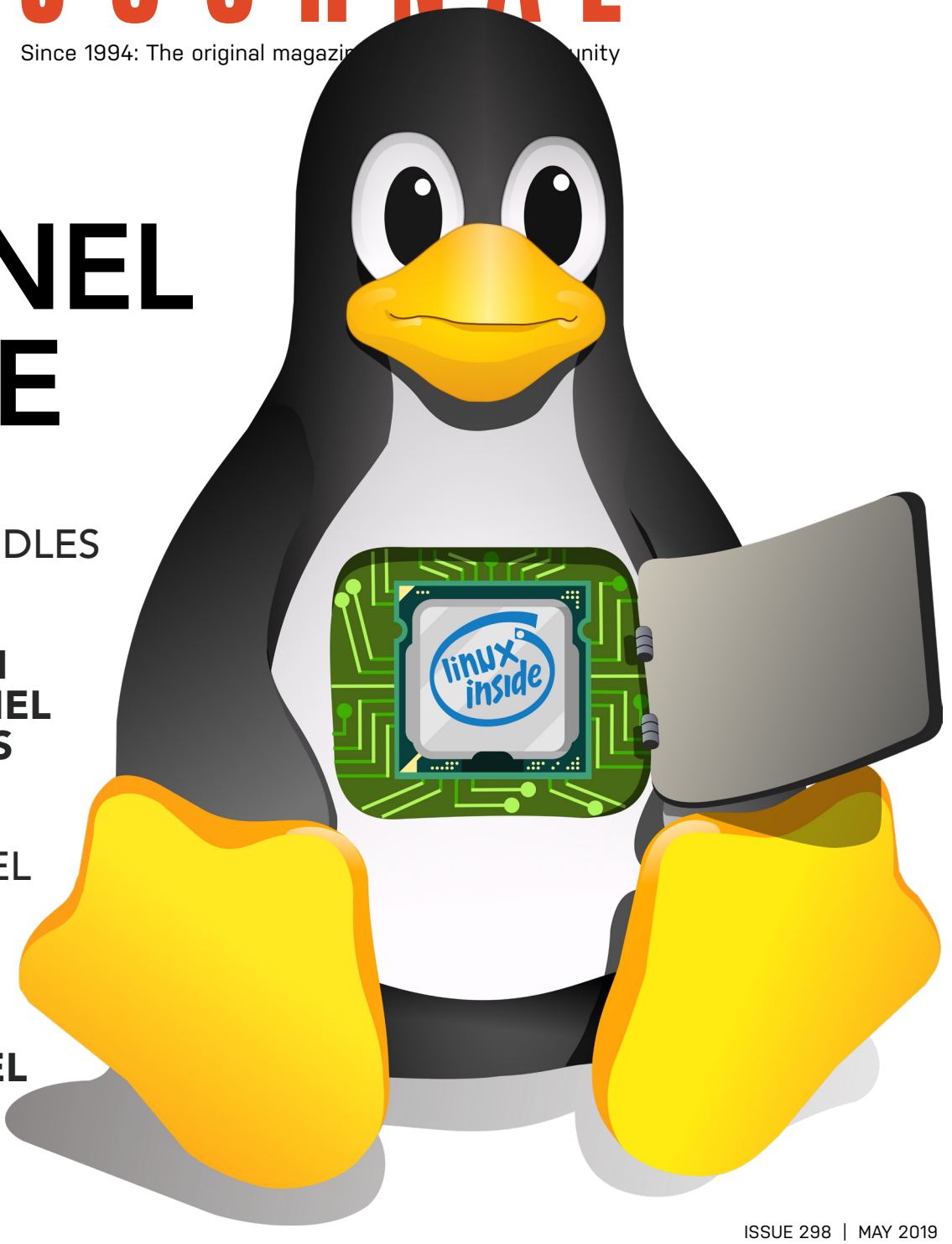
Open Source and Human Genomes

Tighten Your Code with Mypy

LINUX JOURNAL

Since 1994: The original magazi unity

THE KERNEL ISSUE



HOW THE KERNEL HANDLES MEMORY

A TALK WITH THREE KERNEL DEVELOPERS

DEBUGGING LINUX KERNEL PANICS

HOW TO BUILD YOUR OWN KERNEL

ISSUE 298 | MAY 2019
www.linuxjournal.com

CONTENTS | MAY 2019 ISSUE 298

CONTENTS

LINUX JOURNAL (ISSN 1075-3583) is published monthly by Linux Journal, LLC. Subscription-related correspondence may be sent to 9597 Jones Rd, #331, Houston, TX 77065 USA. Subscription rate is $34.50/year. Subscriptions start with the next issue.

ARTICLES

AT YOUR SERVICE

SUBSCRIPTIONS: *Linux Journal* is available as a digital magazine, in PDF, EPUB and MOBI formats. Renewing your subscription, changing your email address for issue delivery, paying your invoice, viewing your account details or other subscription inquiries can be done instantly online: https://www.linuxjournal.com/subs. Email us at subs@linuxjournal.com or reach us via postal mail at *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Please remember to include your complete name and address when contacting us.

ACCESSING THE DIGITAL ARCHIVE: Your monthly download notifications will have links to the different formats and to the digital archive. To access the digital archive at any time, log in at https://www.linuxjournal.com/digital.

LETTERS TO THE EDITOR: We welcome your letters and encourage you to submit them at https://www.linuxjournal.com/contact or mail them to *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Letters may be edited for space and clarity.

SPONSORSHIP: We take digital privacy and digital responsibility seriously. We've wiped off all old advertising from *Linux Journal* and are starting with a clean slate. Ads we feature will no longer be of the spying kind you find on most sites, generally called "adtech". The one form of advertising we have brought back is sponsorship. That's where advertisers support *Linux Journal* because they like what we do and want to reach our readers in general. At their best, ads in a publication and on a site like *Linux Journal* provide useful information as well as financial support. There is symbiosis there. For further information, email: sponsorship@linuxjournal.com or call +1-360-890-6285.

WRITING FOR US: We always are looking for contributed articles, tutorials and real-world stories for the magazine. An author's guide, a list of topics and due dates can be found online: https://www.linuxjournal.com/author.

NEWSLETTERS: Receive late-breaking news, technical tips and tricks, an inside look at upcoming issues and links to in-depth stories featured on https://www.linuxjournal.com. Subscribe for free today: https://www.linuxjournal.com/enewsletters.

LINUX JOURNAL

EDITOR IN CHIEF: Doc Searls, doc@linuxjournal.com

EXECUTIVE EDITOR: Jill Franklin, jill@linuxjournal.com

DEPUTY EDITOR: Bryan Lunduke, bryan@lunduke.com

TECH EDITOR: Kyle Rankin, lj@greenfly.net

ASSOCIATE EDITOR: Shawn Powers, shawn@linuxjournal.com

EDITOR AT LARGE: Petros Koutoupis, petros@linux.com

CONTRIBUTING EDITOR: Zack Brown, zacharyb@gmail.com

SENIOR COLUMNIST: Reuven Lerner, reuven@lerner.co.il

SENIOR COLUMNIST: Dave Taylor, taylor@linuxjournal.com

PUBLISHER: Carlie Fairchild, publisher@linuxjournal.com

ASSOCIATE PUBLISHER: Mark Irgang, mark@linuxjournal.com

DIRECTOR OF DIGITAL EXPERIENCE:
Katherine Druckman, webmistress@linuxjournal.com

DIRECTOR OF SALES: Danna Vedder, danna@linuxjournal.com

GRAPHIC DESIGNER: Garrick Antikajian, garrick@linuxjournal.com

COVER IMAGE: Carty Sewell

ACCOUNTANT: Candy Beauchamp, acct@linuxjournal.com

Linux Journal is published by, and is a registered trade name of, Linux Journal, LLC. 4643 S. Ulster St. Ste 1120 Denver, CO 80237

SUBSCRIPTIONS
E-MAIL: subs@inuxjournal.com
URL: www.linuxjournal.com/subscribe
Mail: 9597 Jones Rd, #331, Houston, TX 77065

SPONSORSHIPS
E-MAIL: sponsorship@linuxjournal.com
Contact: Director of Sales Danna Vedder
Phone: +1-360-890-6285

THE KERNEL ISSUE

By Bryan Lunduke

Bryan Lunduke is a former Software Tester, former Programmer, former VP of Technology, former Linux Marketing Guy (tm), former openSUSE Board Member...and current Deputy Editor of *Linux Journal* as well as host of the (aptly named) *Lunduke Show*.

How much do you know about your kernel? Like *really* know?

Considering how critically important the Linux kernel is to the world—and, perhaps just as important, to our own personal computers and gadgets—it's rather amazing how little most people actually know about it.

There might as well be magical hamsters in there, pushing 1s and 0s around with their enchanted hamster gloves of computing power. How do kernels (in a general sense) actually work, anyway? How does one sit down and debug a specific Linux kernel issue? How does a kernel allocate and work with the memory in your computer? Those are questions most of us never need to ask—because Linux works.

Me, personally? Never submitted a single patch to the kernel. Not one.

I mean, sure. I've looked at little snippets of Linux kernel source code—mostly out of idle curiosity or to investigate a topic for a story. And I've compiled the kernel plenty of times to get one hardware driver or feature working. But, even so, my knowledge of the inner-workings of the kernel is mostly limited to "Linux power user" level.

So, it's time for a little kernel boot camp in this issue of *Linux Journal* to get a bit more up to speed.

Let's start with the basics. What is a kernel, and how, exactly, does a person go about making a brand-new one? Like...from scratch.

Linux Journal Editor at Large Petros Koutoupis previously has walked us through building a complete Linux distribution (starting from the very basics—see Part I and Part II). Now he does the same thing, but this time for building a brand-new kernel.

What tools are needed? What code must be written? Petros provides a step-by-step rundown of kernel building. In the end, you'll have a fully functional kernel (well, functional enough to boot a computer, at any rate) that you can build on further. Plus, you'll have a better understanding of how kernels actually work, which is pretty darn cool.

Moving back to Linux land, Frank Edwards gives a rundown on how the kernel handles memory: how virtual memory works and is structured, how the kernel reports memory usage and information to userland applications and the like. If you've ever wondered how the memory in your system is structured and interacted with by the applications and the kernel, give that a read.

Now that you know the basics of how to build a kernel, and a primer on how memory is used, let's turn to something directly practical for Linux developers and pro users: debugging Linux kernel panics.

Let's say, hypothetically, your machine has a kernel panic. Sure, they're rare, but they happen! But, but, why do they happen? How can you dig in and figure out the cause behind such a catastrophic event?

We bring Petros Koutoupis back in to give a detailed primer and how-to on doing exactly that. Hopefully, you never need to debug a kernel panic. But, just in case, best be prepared. (In the words of a famously pixelated old guy living in a cave, "It's dangerous to go alone! Take this.")

All of this information is great—detailed, technical and nerdy as can be (in the best possible way).

But, let's get a bit higher-level for a moment. What is being a kernel developer actually like?

What gets them started down the kernel programming path? What does an average day in the life of a kernel developer look like? What are their pet peeves about Linux (every developer on every project has complaints about it)?

To answer those questions, I sat down with prominent kernel developers from three of the most active companies in the Linux world: Red Hat, SUSE and Intel. (Since we had all three of those companies represented, this seemed like a good chance to talk about how they interact with other kernel developers working at other companies—often competitors.)

In the end, after reading all of the articles in the pages to follow, maybe you'll be inspired to take your first steps into the world of Linux kernel work. Or, heck, maybe you won't. But, either way, you'll hopefully have a deeper understanding of how Linux (and, by extension, your own computer) works.

Which is empowering. And awesome. And the Linux-y way. ■

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

LINUX JOURNAL

Join the Open-Source Crusade

You subscription includes:

- 12 monthly digital issues
- Fully searchable access to our entire archive (nearly 300 issues)
- Bonus ebook, Sys Admin Fundamentals sent with your paid order

Subscribe.LinuxJournal.com

We Need to Save What Made Linux and FOSS Possible

If we take freedom and openness for granted, we'll lose both. That's already happening, and we need to fight back. The question is how.

By Doc Searls

I am haunted by this passage in a letter we got from reader Alan E. Davis (the full text is in our Letters section):

> ...the real reason for this letter comes from my realization—in seeking online help—that the Linux Documentation Project is dead, and that the Linuxprinting.org project—now taken over by open printing, I think, is far from functioning well. Linux has been transformed into containers, and embedded systems. These and other such projects were the heart and soul of the Free Software movement, and I do not want for them to be gone!

This is the kind of thing Bradley Kuhn (of the Software Freedom Conservancy) lamented in his talk at Freenode.live

Doc Searls is a veteran journalist, author and part-time academic who spent more than two decades elsewhere on the *Linux Journal* masthead before becoming Editor in Chief when the magazine was reborn in January 2018. His two books are *The Cluetrain Manifesto*, which he co-wrote for Basic Books in 2000 and updated in 2010, and *The Intention Economy: When Customers Take Charge*, which he wrote for Harvard Business Review Press in 2012. On the academic front, Doc runs ProjectVRM, hosted at Harvard's Berkman Klein Center for Internet and Society, where he served as a fellow from 2006–2010. He was also a visiting scholar at NYU's graduate school of journalism from 2012–2014, and he has been a fellow at UC Santa Barbara's Center for Information Technology and Society since 2006, studying the internet as a form of infrastructure.

last year. So did Kyle Rankin in his talk at the same event (video, slides and later, an *LJ* article). In an earlier conversation on the same stage (it was a helluva show), Simon Phipps (of the Open Source Initiative) and I had our own lamentations.

We all said it has become too easy to take Linux and FOSS for granted, and the risks of doing that were dire. Some specifics:

- **We collaborate inside proprietary environments**, such as Slack and Google Hangouts. Most of the chat and messaging systems in use today are also proprietary and closed. So are most video-conferencing systems and the codecs they use.

- **Many Linux and FOSS geeks today use Linux only professionally.** Most of their personal work is on proprietary Apple and Microsoft gear. Many use Windows or macOS boxes in presentations about FOSS topics.

- **We're not modeling our values.** Bradley sourced this line from Benjamin Mako Hill: "The use of nonfree tools sends an unacceptable message…'Software freedom is important for you as users', developers seem to say, 'but not for us'. Such behavior undermines the basic effectiveness of the strong ethical commitment at the heart of the free software movement."

- **We've allowed foundational ideas to collapse.** We've gone along with complicating the web, no longer respecting the simplicities in HTTP and HTML, which allowed the web to work in the first place. For example, we hardly still design for what Bradley calls "progressive enhancement and graceful degradation". We see this failure in the web development world, which now depends almost utterly on JavaScript, most of which is proprietary and downloaded constantly on the fly to run in browsers.

- **We are also forgetting (or perhaps never learned) how a reciprocal license, such as the GPL, can keep a project alive and a community together.** Simon blames SourceForge's failures on a decision to replace its original free (GPL-licensed) software base with a proprietary one. And now,

even though we have Git, he says too many of us don't know the difference between Git and GitHub, or that GitHub runs proprietary JavaScript executed in our browsers.

There were signs this was coming in 2002, when I wrote "A Tale of Three Cultures". I'll unpack those a bit:

- **Geeks** at the time were busy inventing the world's basic software building materials. They operated in a culture that valued freedom, openness and maximized usefulness to everybody and everything. They also had a strong sense that they were winning the fight for freedom and openness in software development and product design. In geek slang, they said they were at "GandhiCon 3". (The context is a Mohandas Gandhi one-liner: "First they ignore you. Then they laugh at you. Then they fight you. Then you win.")

- **Hollywood** as a label stood for all that is proprietary about business. I chose that label because the biggest public fight at the time was over copyright, and Hollywood was (and remains) the embodiment of copyright maximalism. Larry Lessig, who with Aaron Swartz and others had recently minted Creative Commons, characterized the fight as Silicon Valley vs. Hollywood, and Northern vs. Southern California.

- **Embedded developers** were what I called "purely technical…pre-Net, pre-UNIX and maybe even pre-cultural", with concerns that were "utterly practical". In other words, *not* about free software, open source or Linux—beyond its utilitarian value. I wrote that after attending the Embedded Systems Conference that Rich Lehrbaum wrote about for *Linux Journal*, here. (That may be the only surviving record of the conference on the web.)

What I didn't see back then was that Hollywood and embedded would become pretty much the same thing: business as usual. That happened because it was too easy for too many developers to build proprietary and closed stuff, heads down, in utterly practical ways, usually for what amounted to embedded purposes, on top of Linux and FOSS foundations, with little respect for the virtues embodied in those foundations. And by now, we've built a lot of it. One might even

argue that most of the Linux deployed in the world today is embedded inside proprietary and closed devices.

So the question is *What should we do now?*

From my notes, here are some things Bradley, Kyle, Simon and others said at Freenode.live. It's not all verbatim, but close enough:

- "Having real-time chat is absolutely essential to the advancement of free software."
- "We're the resistance now." "We need to create mass movement."
- "Volunteer to write free and open code, to participate in communities."
- "If you didn't live the history, learn from those who did."
- "If you did learn from history, teach those who need to know it. Respectfully."
- "Be patient. Remember that the tortoise won not only because it was patient, but because it ignored insult, ridicule and dismissal."
- "Model your values. Use free software and hardware."
- "Remember always how 'the rights to copy, share, modify, redistribute and improve software' are fundamental rights that matter to people."
- "Work to convince developers that their software freedom matters."

That's all necessary, but not sufficient. We need something more. Something big.

I suggest we pick a fight. Because fights raise emotions and have goals.

I just ran a playoff between many different fights on many tabs in a browser. The winner—the last tab standing—is "The Era of General Purpose Computers Is

Ending", by Michael Feldman in The Next Platform website. It's a sad bookend to the history of a losing fight that Cory Doctorow forecast in 2011 with "Lockdown: the coming war on general-purpose computing" and a year later in "The Coming Civil War over General Purpose Computing". Read all three.

I chose general-purpose computing as the winning fight—the one most worth having—because we wouldn't have Linux, free software or open source today if there weren't general-purpose computers to develop and use them on. General-purpose computing is the goose that laid all our golden eggs. The fight is to keep it alive. ■

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

LETTERS

Thanks for the Ansible Articles

I'm Michael, a systems administrator in Waterloo, Canada. I was interested in Ansible and tried to find some good articles or lectures on the internet, but unfortunately, most of them just explain all the functions and are hard for me to understand. When I read Shawn Powers' article in *Linux Journal*, it was really interesting and understandable and easy to understand. So, thank you for that.

—Michael

Note: if you're interested in Ansible, you can read Shawn Powers' series on our website: "Ansible: the Automation Framework That Thinks Like a Sysadmin", "Ansible: Making Things Happen", "Ansible, Part III: Playbooks" and "Ansible, Part IV: Putting It All Together".—Ed.

Rankin, Searls and Taylor

Doc Searls' editorials, Kyle Rankin's "Hack and /" and Dave Taylor's "Work the Shell" keep me subscribing to *LJ*. Sure the specialty articles are great, but I am a power user, not a sysadmin or "IT guy". Keep feeding me tips on how to get more from the Linux command line! Catching up on the January issue, I particularly appreciated seeing the options on `sort` and `uniq` that had escaped my notice (see Kyle Rankin's "Back to Basics: sort and uniq".) Nice work...all!

—Richard

Historical Errors

According to my friend, Robert Wachtel, Dave Taylor's article from the April 2019 issue "Back in the Day: UNIX, Minix and Linux" contains some inaccuracies. Specifically:

> Interesting but inaccurate regarding PARC and Doug Engelbart. Engelbart and his group at SRI created the mouse and windows before PARC was founded.

See The Mother of All Demos and PARC (company), on Wikipedia.

—Roger

Dave Taylor replies: Entirely possible I mis-remembered my timeline, but I do know that Engelbart was working at SRI when we met, and I heard him talk about his "mouse". If I suggested that he was at PARC, that was my mistake, although the PARC systems definitely utilized that mouse device!

A Matter, Perhaps, of Philosophy

I have recently subscribed to *LJ*. When *LJ* first come into being, I was unable to subscribe, literally; I was a teacher on an undeveloped island, with an embarrassingly low salary. I actually began to use GNU/Linux because I could not afford to buy Multiedit, which would have granted access to the documentation. I needed to be able to type diacritics, and without access to the documentation, I couldn't figure out how to do so. I wrote a request letter to the Free Software Foundation, begging for a free text editor. Little would I have suspected how their gift to me—13 3-1/2″ diskettes of GNU software ported to Windows—would change my life. Emacs is the self-documenting editor, and the documentation is at one's fingertips at all times. My project was a lexicon; text tools like `grep`, a functioning `sort`, and `ptx` were extremely useful.

I started out on the wrong side of the Free Beer vs. Freedom divide. Maybe not, though, because the availability of tools is extremely important in the struggle for freedom, at all levels. I was a science teacher, so this was critically important to me. I started reading the GNU's Bulletin, wearing out each issue as they arrived when I was able to travel to a less remote island. I learned of two free operating systems through GNU's Bulletin: FreeBSD and "Linux". It came to pass that I was able to download a copy of Slackware and started using it. I never looked back.

I tell this tale to accentuate the liberating nature of Free Software. The tool-lending library in my city makes a range of useful hardware available to anyone with a

library card, without charge. I cannot explain the passion that these developments awaken within me.

I grew up in a relatively well-to-do environment, at almost every level. Yet, I ended up living on an island, on which cash has a minimal importance except to buy those things that were introduced by the benevolent other-world empire. I bring this up because I now am living back in my own "world" where I am constantly being reminded of the preeminence of money.

Linux Journal represented to me an attempt to grow a business for profit through association with an ecosystem that is free—not only in the "open source" manner of thinking, but as something that could be used even by those who were unable to afford, say, a copy of Word or Word Perfect costing several hundred dollars—not to mention the superior quality of the tools that Free Software has made available.

Yet I have scoured every *Linux Journal* I could get hold of. They were sold in some bookshops, and I occasionally could allocate a few dollars to purchase a copy. I did subscribe, but could not pay.

I do not and would not resent the efforts of another to feed himself and his family through publishing. I get it that *Linux Journal* was not a hugely successful capitalistic enterprise. I don't mind paying for a subscription for a year. (Heck, even the libraries around here do not carry it.) *LJ* is still the best of the Linux magazines. But something has happened, and that something—whatever it is—is reflected in the manner of content that is offered within its covers.

Today, I am writing because I just spent borrowed money to purchase a printer. It is one of the new breed that promises (and to some extent seems to deliver) a new paradigm—ink tanks. My old printer was still on warranty, but I have been using unblessed ink, and to take it to the repair shop for promised repair at the authorized service center will probably require me to purchase a full set of ink cartridges. The cheapest I have found costs about 60.00. I have been able to print for less than 20.00 a year in ink with oversize ink cartridges made in China, with quality that is good

enough, if not absolutely matched in color. Now I have received an error message: "Ink Absorber Pad Full". The service department had advised me over the phone to purge the cartridges, emptying them of ink. I'll say this, the drivers were easy to install on an Arch Linux system, or Manjaro.

Cutting to the chase, this new printer is a different beast. The drivers are more difficult to install, and the scanner does not work as it should. The settings in CUPS are few, compared with the many I have seen in the pictures of Windows' settings windows. It does interesting things. It is a new model, and the manufacturer has provided it with an email address: all I need is to send a document to that email address, and it will be printed. We'll see. I have access to it from my Android phone, and presumably a tablet, including nozzle cleaning and etc.

It's on me for not shopping more specifically for a Linux-friendly printer. Are there any? Really? But the real reason for this letter comes from my realization—in seeking online help—that the Linux Documentation Project is dead, and that the Linuxprinting.org project—now taken over by open printing, I think—is far from functioning well. Linux has been transformed into containers and embedded systems. These and other such projects were the heart and soul of the Free Software movement, and I do not want for them to be gone!

The spirit of free software is under threat in this perilous time. Microsoft is now embracing Linux like a giant anaconda, seeking to squeeze more profit.

I don't know what to suggest, but I would like to see more sensitivity to those people who are still floundering, confused about installing printers, or unable—like my good friend who has for years struggled to install and use Linux has recently experienced—to get Secure Boot turned off on a Windows 10 laptop, several years old, to install some distro of Linux.

Is there some contribution that *Linux Journal* can make to the community of users who are being worn down by the corporate flim flam? I have tried for years to advocate for GNU/Linux (with an affectionate, gentle touch on "GNU/"). GNU/Linux

has changed my life. I have failed to convince those teachers around me—in schools where Windows and Apple software are provided for by federal grants. A few students picked it up. There is some really important work still to be done.

This all being said, I look forward to scouring every issue of *LJ* over the coming year. I appreciate the new and enthusiastic leadership of Doc Searls and also Bryan Lunduke's wild bits. There are still some of us who actually are down here on Earth, storing our bits on our own hardware, and struggling with the efforts of the corporate world not only to ruin our political lives and steal our eyeballs, but also to force us to buy the bill of goods they are wont to sell.

—Alan Davis

Doc Searls replies: Thanks, Alan. Your letter hit home for me in a big way, and I've answered with my From the Editor column this month.

Great Article

Regarding Doc Searls' article "The Kids Take Over" in the April 2019 issue, I wish I was eight again and in school with that KidOYO program. That educational program is stunning. And the part I like the most is no one is left behind. Thank you for finding and sharing it. We need more people to think the way those folks in New York are thinking.

—Bob Getsla

SEND *LJ* A LETTER *We'd love to hear your feedback on the magazine and specific articles. Please write us here or send email to ljeditor@linuxjournal.com.*

PHOTOS *Send your Linux-related photos to ljeditor@linuxjournal.com, and we'll publish the best ones here.*

Visualizing Science with ParaView

I'd like to introduce one of the more popular tools used for visualizing data within several scientific disciplines: ParaView. ParaView started as a joint project between Kitware, Inc., and Los Alamos National Laboratory back in 2000. The first public release was version 0.6, which came out in 2002. Since then, ParaView has become one of the most popular visualization packages for visualizing large data sets.

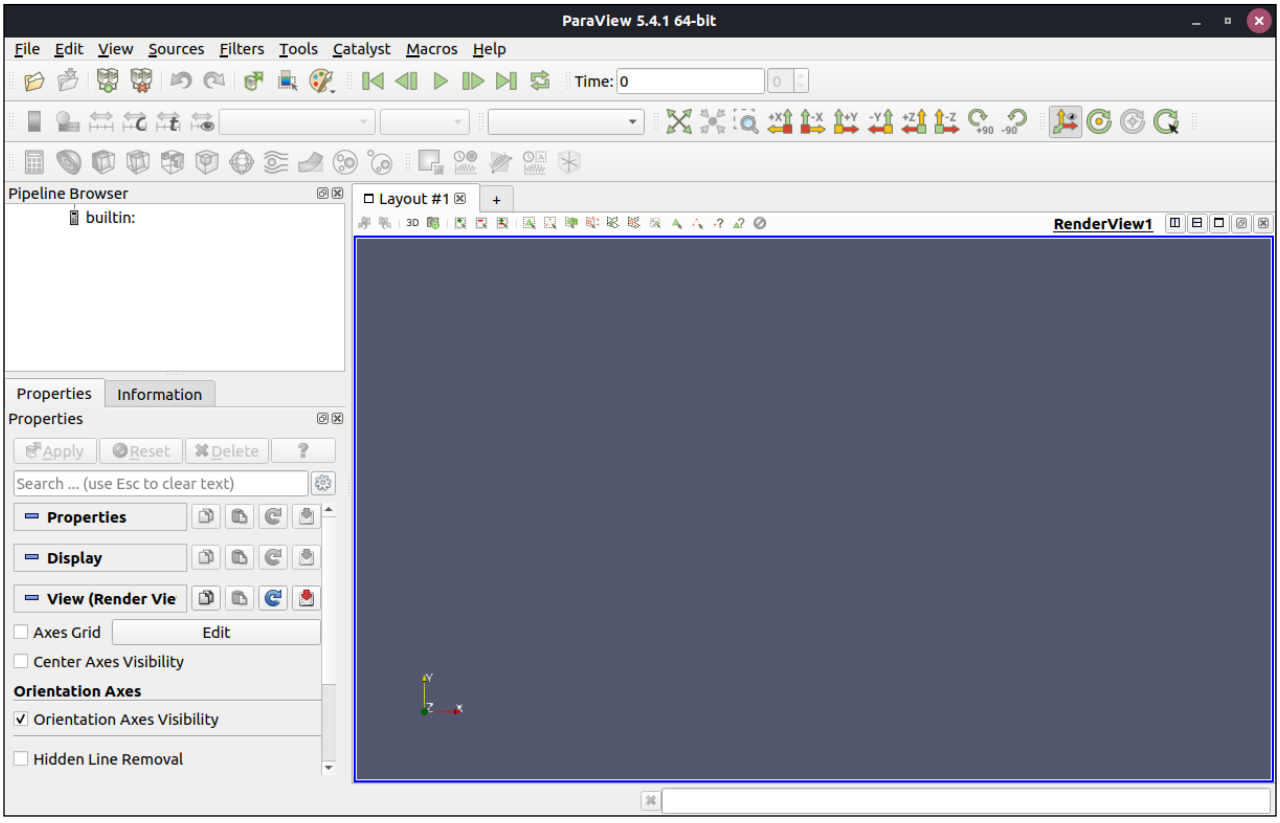


Figure 1. When you first start ParaView, you'll see a new, empty layout to start your visualization.

Because it's open source, it should be available in most, if not all, package repository systems. For example, in Debian-based distributions, you should be able to install it with the command:

```
sudo apt-get install paraview
```

Starting it the first time should give you an empty workspace, ready for you to get to work.

Two major parts populate the bulk of the window. The right-hand side is the main display pane where the visualization will appear. The left-hand pane shows the list of objects being visualized, along with their properties. At the top, there is a toolbar of

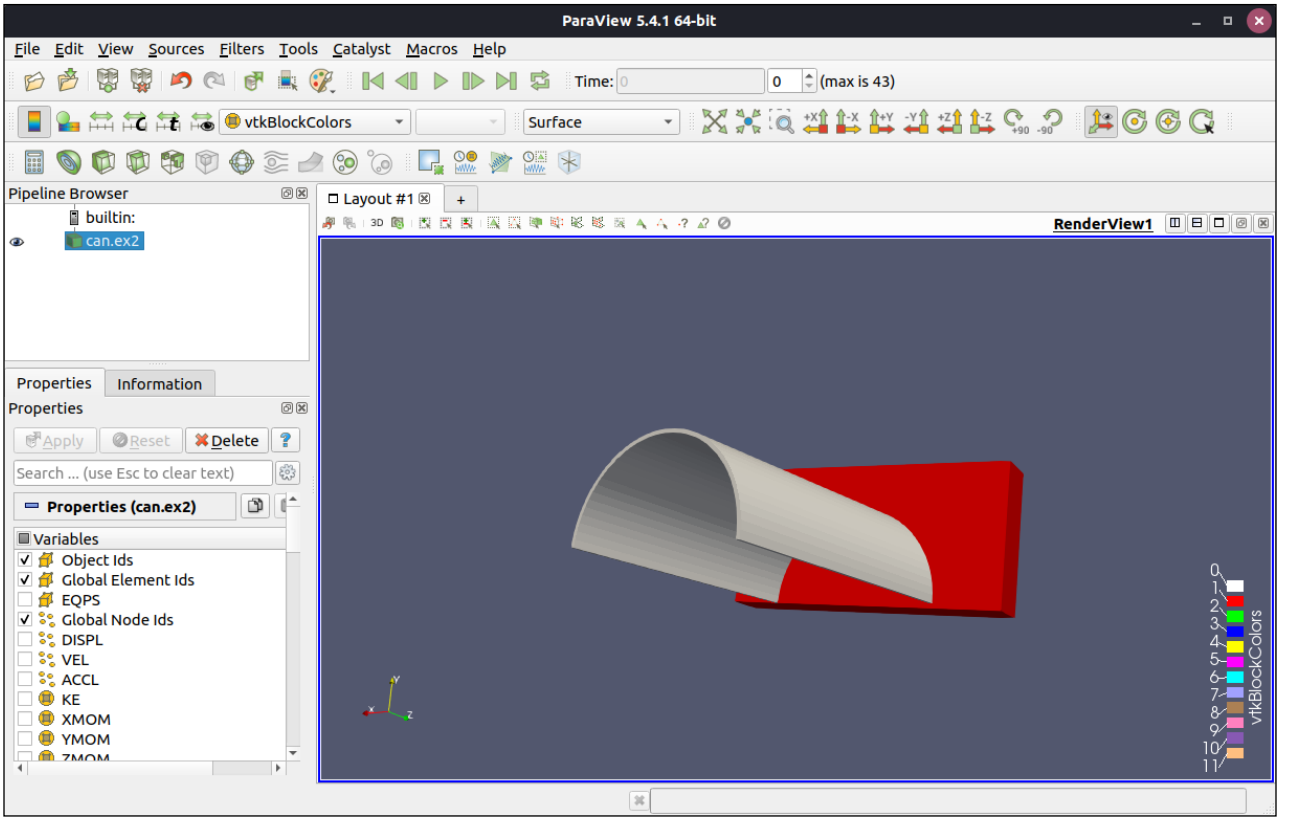


Figure 2. The data in the sample file can.ex2 renders as a half cylinder attached to a rectangle on the end.

the common functions in ParaView.

To play with ParaView, you'll need some data. If you don't have any data of your own to use, you can grab some data provided as part of the ParaView Tutorial. More documentation and sample scripts are also available there.

Let's assume you're going to use the sample data as you learn how to use ParaView. To load the data, click File→Open, and navigate to where you unpacked the sample data.

While you're here, take a quick look at the list of all of the file types ParaView supports. For example, you can load the data stored in the file can.ex2. You won't see anything displayed right away. In the bottom part of the left-hand side pane, you should see the properties for the newly loaded data file. For now, you can just accept the defaults and

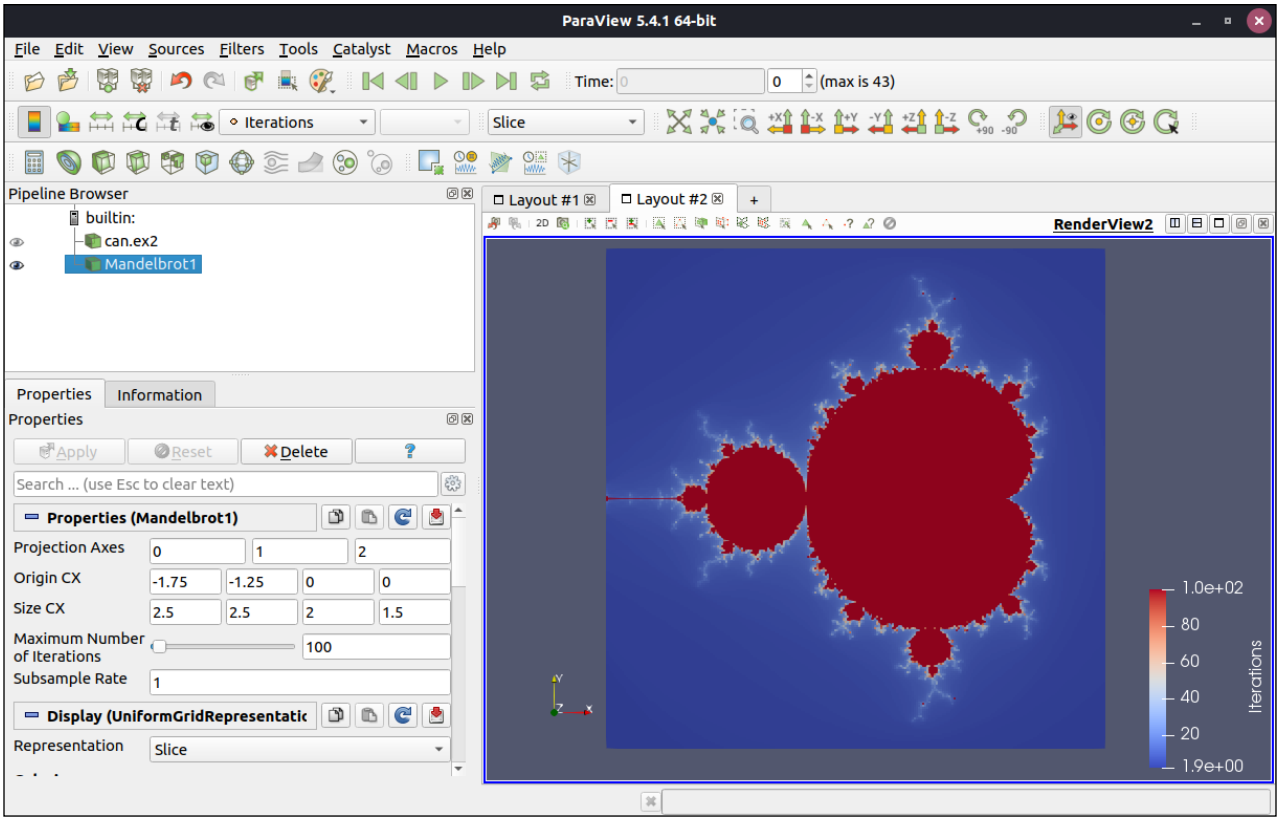


Figure 3. You can add lots of different objects to your visualization, even a Mandelbrot set.

click the apply button. You then should see the data visualized in the main pane.

Clicking and dragging on the image allows you to rotate the view, so you can see the entire object from various angles.

Along with visualizing data, ParaView includes a number of basic shapes you can use to build up structures within your visualization. Clicking the Sources menu item provides a fairly lengthy drop-down list of structures. And, you even can add more complicated structures (like the Mandelbrot set) to your visualization.

This could be handy if you have some basic geometric structure or an image that you want to use as a backdrop to your data visualization.

If the data you're visualizing is more traditional (for example, if the data comes from measurements), ParaView provides actual data analysis tools to complement

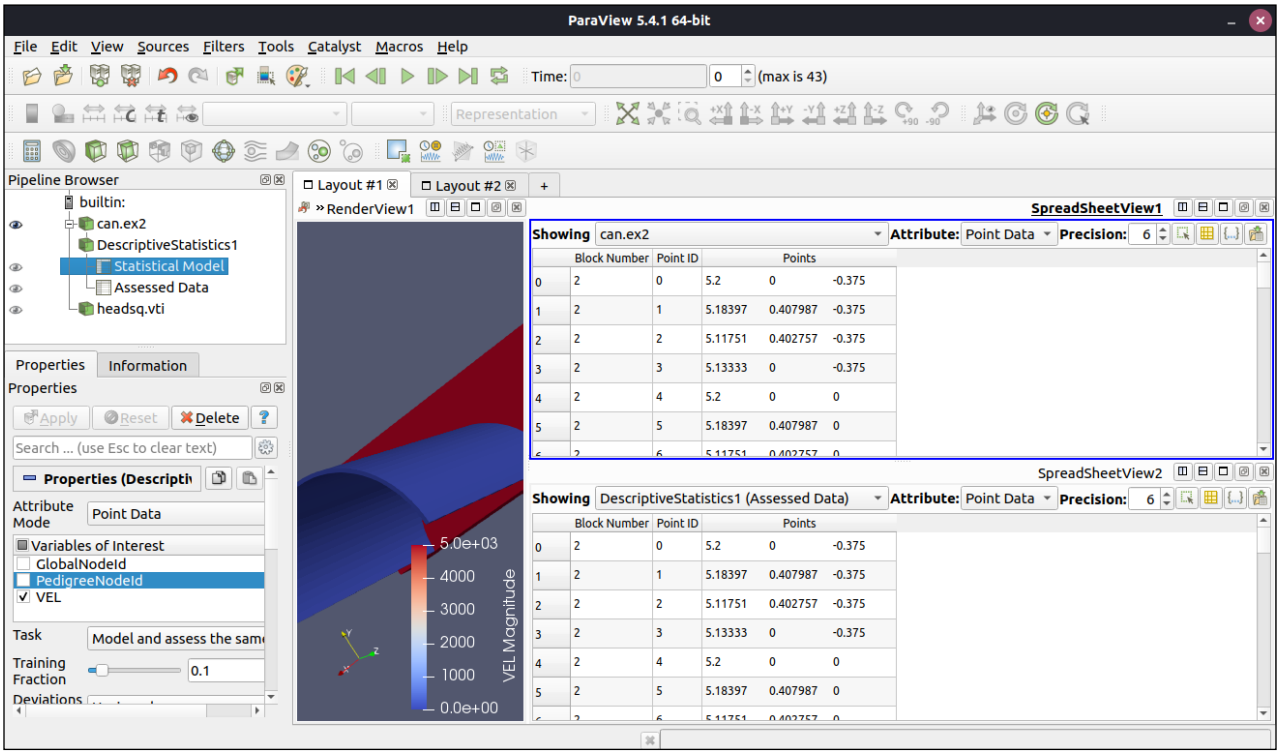


Figure 4. You can add statistical analysis to your pipeline of visualization steps in your analysis.

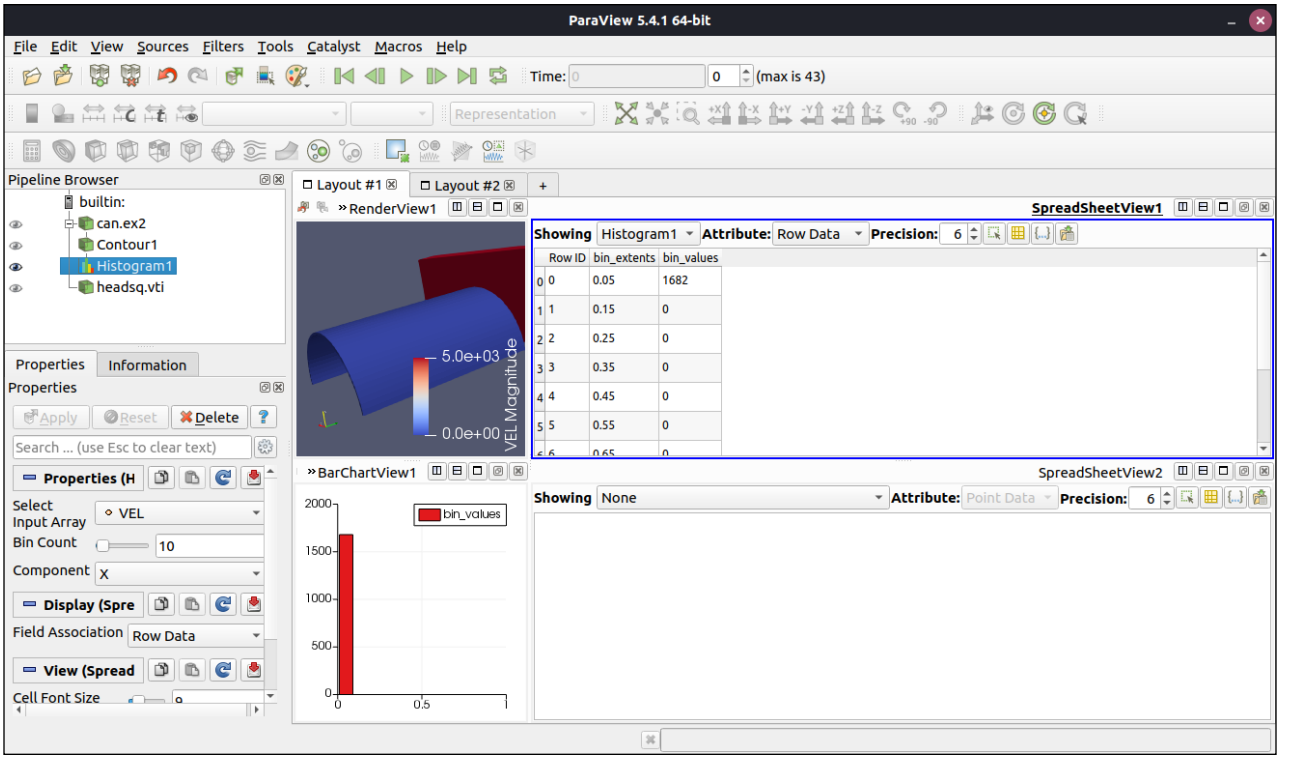


Figure 5. You even can do histograms of the data being visualized.

the visualization tools. For example, clicking the Filters→Statistics menu item provides a drop-down list of statistical functions. Clicking the "Descriptive Statistics" option adds a new entry in the "Pipeline Browser" where you can set the options for the statistical analysis.

This opens a new pane where you can play with the data a bit more directly. This particular data set is not very interesting, so descriptive statistics aren't very useful in this specific case.

You also can do more detailed data analysis by clicking the Filters→Data Analysis menu item. For example, clicking the histogram entry gives you a new pane displaying a histogram plot.

You also can do things like calculate quartiles or replot interpolated and analyzed data.

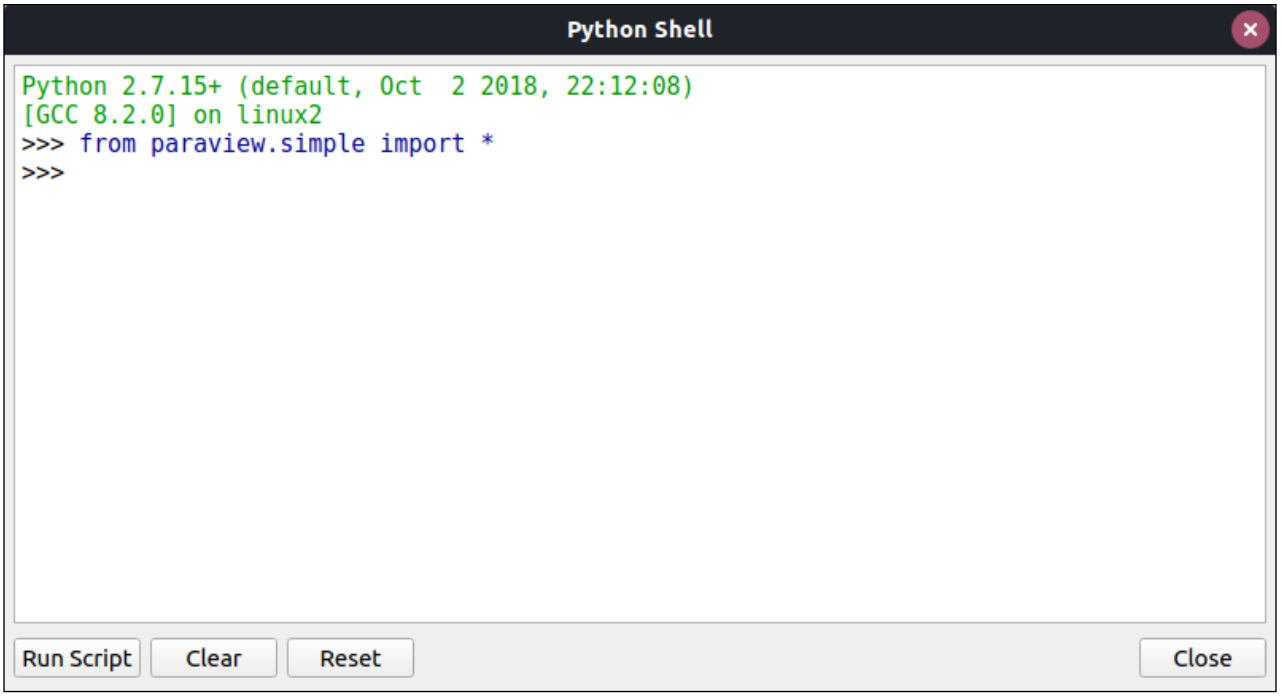


Figure 6. Within ParaView, you have access to a Python shell where you can interact with the ParaView analysis tools directly.

For repeated visualization, you probably won't want to go through all of the required steps every time. ParaView includes a Python engine, so you can write a Python script that can run repeated processing steps easily. This also means you can script behavior that can be processed when the GUI is not active. This comes in handy when you're running larger data analysis jobs on high-performance clusters remotely.

You can work on your Python scripting by clicking Tools→Python Shell. This pops up a new window where you can write and evaluate your Python code directly within ParaView.

Along with writing Python scripts, ParaView has been designed with a plugin architecture. Clicking Tools→Manage Plugins pops up a new window where you can select which plugins are loaded and active.

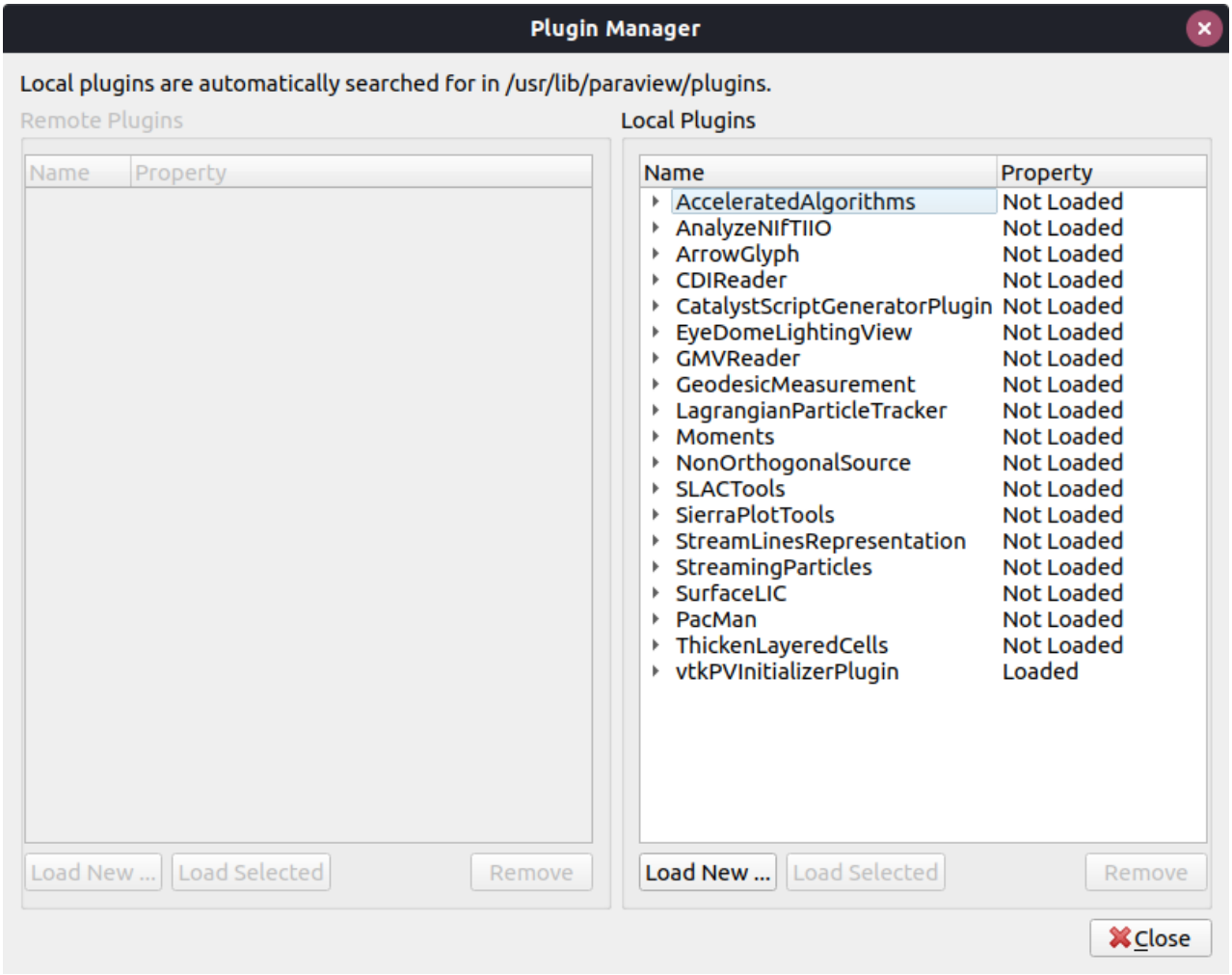


Figure 7. ParaView supports plugins, but it's up to you to select which ones are active and loaded for use in a current session.

If you're in the middle of some visualization work, you can save the current state of ParaView so that you can pick it up again later. Clicking File→Save State lets you save the current state as a .pvsm (ParaView state) file. You can reload it later by clicking File→Load State.

Once you've finished a visualization, there are a couple options that allow you to generate files that you can use in other software packages. Clicking File→Save Screenshot pops up a new window where you can set options like the image size.

Then a second window will open where you can set the image filename.

The File→Export Scene menu item gives you a second option for saving your results. In this case, you can save your results in other file formats, such as PostScript or PDF. If your visualization includes an animation, click File→Save Animation to save it.

If you're working with large or complicated data sets, I recommend making the move to ParaView as your main visualization tool.

—Joey Bernard

Patreon and *Linux Journal*

PATREON

Together with the help of *Linux Journal* supporters and subscribers, we can offer trusted reporting for the world of open-source today, tomorrow and in the future. To our subscribers, old and new, we sincerely thank you for your continued support. In addition to magazine subscriptions, we are now receiving support from readers via Patreon on our website. *LJ* community members who pledge $20 per month or more will be featured each month in the magazine. A very special thank you this month goes to:

- Appahost.com
- Chris Short
- Christel Dahlskjaer
- David Breakey
- Dr. Stuart Makowski
- Fred
- Henrik Halbritter (Albritter)
- James Mayes
- James Weatherell
- Joe
- Josh Simmons
- LinuxMagic Inc.
- Lorin Ricker
- Taz Brown

BECOME A PATRON

Now also find @linuxjournal on Liberapay. Thank you to our very first Liberapay supporter and the person who gave us this great suggestion: Mostly_Linux.

Reality 2.0: a *Linux Journal* Podcast

Join us each week as Doc Searls and Katherine Druckman navigate the realities of the new digital world: https://www.linuxjournal.com/podcast.

Signing Git Commits

Protect your code commits from malicious changes by GPG-signing them.

Often when people talk about GPG, they focus on encryption—GPG's ability to protect a file or message so that only someone who has the appropriate private key can read it. Yet, one of the most important functions GPG offers is *signing*. Where encryption protects a file or message so that only the intended recipient can decrypt and read it, GPG signing proves that the message was sent by the sender (whomever has control over the private key used to sign) and has not been altered in any way from what the sender wrote.

Without GPG signing, you could receive encrypted email that only you could open, but you wouldn't be able to prove that it was from the sender. But, GPG signing has applications far beyond email. If you use a modern Linux distribution, it uses GPG signatures on all of its packages, so you can be sure that any software you install from the distribution hasn't been altered to add malicious code after it was packaged. Some distributions even GPG-sign their ISO install files as a stronger form of MD5sum or SHA256sum to verify not only that the large ISO downloaded correctly (MD5 or SHA256 can do that), but also that the particular ISO you are downloading from some random mirror is the *same* ISO that the distribution created. A mirror could change the file and generate new MD5sums, and you may not notice, but it couldn't generate valid GPG signatures, as that would require access to the distribution's signing key.

Why Sign Git Commits

As useful as signing packages and ISOs is, an even more important use of GPG signing is in signing Git commits. When you sign a Git commit, you can prove that the code you submitted came from you and wasn't altered while you were transferring it. You also can prove that you submitted the code and

not someone else.

Being able to prove who wrote a snippet of code isn't so you know who to blame for bugs so the person can't squirm out of it. Signing Git commits is important because in this age of malicious code and back doors, it helps protect you from an attacker who might otherwise inject malicious code into your codebase. It also helps discourage untrustworthy developers from adding their own back doors to the code, because once it's discovered, the bad code will be traced to them.

How to Sign Git Commits

The simplest way to sign Git commits is by adding the `-S` option to the `git commit` command. First, figure out your GPG key ID with:

```
gpg --list-secret-keys --keyid-format LONG
sec# rsa4096/B9EF770D6EFE360F 2019-02-06 [SC]
 ↪[expires: 2021-02-05]
. . .
```

In this case, `B9EF770D6EFE360F` is my long key ID. Why use this and not just my email address associated with my key? In the event you have multiple keys with the same ID, you might end up signing with the wrong key. By specifying the long key ID, you can ensure that you use the right key every time.

Once you know the key ID, add it to the `-S` option when you `git commit`:

```
git commit -S B9EF770D6EFE360F
```

Now when you submit the commit, it will prompt you to unlock your GPG key so it can sign the commit.

Of course, the goal is to sign every commit, and if you had to add this argument every time you committed code, it would be pretty annoying. So instead, add it to your ~/.gitconfig so it signs every time:

```
[user]
  name = Kyle Rankin
  email = kyle.rankin@puri.sm
  signingkey = B9EF770D6EFE360F
[commit]
  gpgsign = true
```

In the `[user]` section of your .gitconfig, after your name and email, add a `signingkey` option, and set it to the same key ID you used for the `-S` argument in `git commit`. Then add a new `[commit]` section, if it doesn't already exist, and add the `gpgsign` option set to `true`. This way, all of your GPG commits will be signed.

The final step once this is all set up, if you use a web-based Git repository like GitLab or GitHub, is for you to go to your shared Git repository, log in to your account, and find the section that lets you upload GPG public keys, so you can add your corresponding GPG public key to your account. This way, when you do sign your commits, the Git repository will be able to verify the signature against your public key and add a handy "Verified" tag that denotes that the commit came from you.

—Kyle Rankin

FOSS Project Spotlight: Bareos, a Cross-Network, Open-Source Backup Solution

Bareos (Backup Archiving Recovery Open Sourced) is a cross-network, open-source backup solution that preserves, archives and recovers data from all major operating systems. The Bareos project started 2010 as a Bacula fork and is now being developed under the AGPLv3 license.

The client/server-based backup solution is actually a set of computer programs (Figure 1) that communicate over the network: the Bareos Director (BD), one or more Storage Dæmons (SD) and the File Dæmons (FD). Due to this modular design, Bareos is scalable—from single computer systems (where all components run on one machine) to large infrastructures with hundreds of computers (even in different geographies).

The director is the central control unit for all other dæmons. It manages the database (catalog), the connected clients, the file sets (they define which data Bareos should back up), the configuration of optional plugins, before and after jobs (programs to be executed before or after a backup job), the storage and media pool, schedules and the backup jobs. Bareos Director runs as a dæmon.

The catalog maintains a record of all backup jobs, saved files and volumes used. Current Bareos versions support PostgreSQL, MySQL and SQLite, with PostgreSQL being the preferred database back end.

The File Dæmon (FD) must be installed on every client machine. It is responsible for the backup as well as the restore process. The FD receives the director's instructions, executes them and transmits the data to the Bareos Storage Dæmon. Bareos offers pre-packed file dæmons for many popular operating systems, such as Linux, FreeBSD,

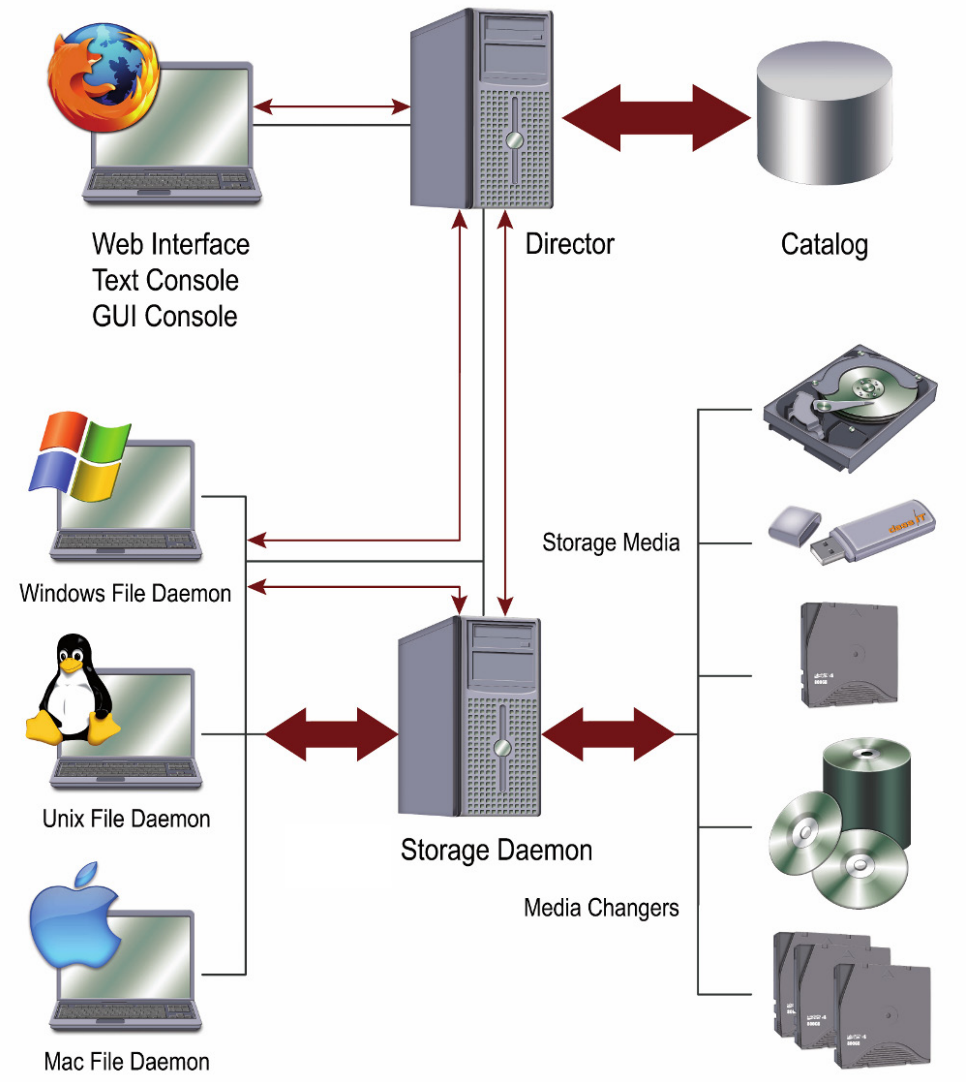


Figure 1. A Typical Bareos Setup: Director (with Database), File Dæmon(s), Storage Dæmon(s) and Backup Media

AIX, HP-UX, Solaris, Windows and macOS. Like the director, the FD runs as a dæmon in the background.

The Storage Dæmon (SD) receives data from one or more File Dæmons (at the director's request). It stores the data (together with the file attributes) on the configured backup medium. Bareos supports various types of backup media, as shown in Figure 1, including disks, tape drives and even cloud storage solutions. During the restore process, the SD is responsible for sending the correct data back to the FD(s). The Storage Dæmon runs as a dæmon on the machine handling the backup device(s).

Backup Jobs

A backup job defines what to back up (`FileSet` directive for the client), when to back up (schedule) and where to back up (for example, on a disk, tape, etc.). Bareos is quite flexible, and you can mix different directives. So you can have different job definitions (resources), backing up different machines, but using the same schedule, the same `FileSet` and even the same backup medium.

The schedule describes what kind of backup (full, incremental or differential) runs on different days of the week or month. If more than one backup job relies on the same schedule, it's possible to set the job priority and tell Bareos which job is supposed to run first. Additionally, there are restore, verify and admin jobs.

bconsole and WebUI

The Bareos configuration is stored in text files. In order to communicate with

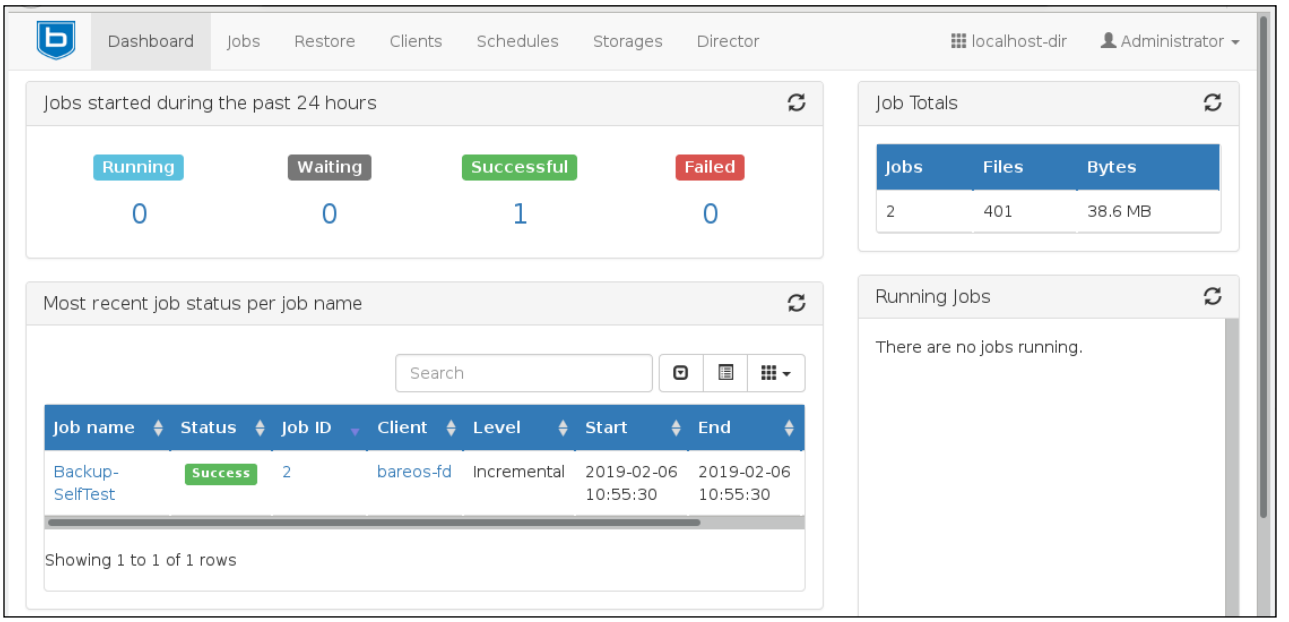


Figure 2. The Bareos WebUI allows users to monitor the backup solution as well as restore their data.

the director, administrators (and other authorized users) can use the command-line tool Bareos Console (`bconsole`). The shell interface allows you to query Bareos' state, determine the status of a particular job, examine the contents of the database and run jobs manually.

You can run `bconsole` basically anywhere on your network—it doesn't have to be the BD machine. Since Bareos Console is a shell interface, it also works via SSH.

The Bareos WebUI (Figure 2) has been part of the backup solution since version 15.2.0. The multilingual web interface can access multiple directors and catalogs. Its main purpose is to monitor the backup software, but it's also possible to start, cancel or rerun jobs. You can use the WebUI to restore files (even to a different client) and browse through a file tree of backup jobs.

Special Features

Bareos values security and safety and supports transport encryption as well as data encryption. The software uses TLS (Transport Layer Security) for all network connections. On top of that, it's possible to encrypt and sign the data on the File Dæmon before the backup is sent to the Storage Dæmon. Encryption and signing are implemented using RSA private keys coupled with self-signed X.509 certificates (PKI, Public Key Infrastructure).

You can extend Bareos' functionality by adding plugins to the director, FD and SD. For example, there are LDAP plugins, plugins to back up and restore various database back ends (PGSQL, MySQL, MSSQL), extensions for GlusterFS and Ceph filesystem, and a VMware plugin for agentless backups of virtual machines running on VMware vSphere. The bpipe plugin is a generic pipe program that simply transmits the data from a specified program to Bareos and back.

Documentation and Support

For more information on Bareos, please have a look at the official documentation. There are also two mailing lists: bareos-users (for users, for help from the community) and bareos-devel (for developers, for discussions on how to modify

the Bareos code). The source code is available in the Bareos GitHub repository.

The Bareos download servers offer packages for all major operating systems (stable Bareos version only). Subscription customers can get access to repositories with maintenance and bug-fix releases. The company Bareos GmbH & Co. KG (located in Cologne, Germany) also offers professional support.

—Heike Jurzik and Maik Aussendorf

News Briefs

Visit LinuxJournal.com for daily news briefs.

- Microsoft has published the code for Windows Calculator and released it on GitHub under the permissive MIT license. *Ars Technica* reports that "The repository shows Calculator's surprisingly long history. Although it is in some regards one of the most modern Windows applications—it's an early adopter of Fluent Design and has been used to showcase a number of design elements—core parts of the codebase date all the way back to 1995."

- Audacity recently released version 2.3.1. This new version restores Linux support, which was missing in the previous version, and also fixes more than 20 bugs and improves Audacity for macOS. For details on all the new features, go here, and see also the release notes.

- Flickr has announced that all CC-licensed images will be protected. According to the Creative Commons article, "all CC-licensed and public domain images on the platform will be protected and exempted from upload limits. This includes images uploaded in the past, as well as those yet to be shared. In effect, this means that CC-licensed images and public domain works will always be free on Flickr for any users to upload and share."

- Purism announces that along with three kill switches, Librem 5 smartphone also will have a new feature called "Lockdown Mode". As far as the kill switches, one is for cameras and microphone, one for WiFi and Bluetooth, and one for cellular baseband. Lockdown Mode goes further and "extends our normal kill switches to provide even more security and privacy". Purism's Chief Security Officer Kyle Rankin writes, "When in Lockdown Mode, in addition to powering off the cameras, microphone, WiFi, Bluetooth and cellular baseband we also cut power to GNSS, IMU, and ambient light and proximity sensors. Lockdown Mode leaves you with a perfectly usable portable computer, just with all tracking sensors and other hardware disabled. If you switch any of the hardware kill switches back

on, the hardware that corresponds to that switch powers on along with GNSS, IMU, and ambient light and proximity sensors."

- Firefox announced its new Firefox Send feature. According to the Mozilla Blog post, "Send is a free encrypted file transfer service that allows users to safely and simply share files from any browser. Additionally, Send will also be available as an Android app in beta later this week." You also can decide when the link expires, select the number of downloads and optionally add a password for more security.

- The FSF awarded seven devices from ThinkPenguin with its Respects Your Freedom (RYF) certification. The devices include "The Penguin Wireless G USB Adapter (TPE-G54USB2), the Penguin USB Desktop Microphone for GNU/Linux (TPE-USBMIC), the Penguin Wireless N Dual-Band PCIe Card (TPE-N300PCIED2), the PCIe Gigabit Ethernet Card Dual Port (TPE-1000MPCIE), the PCI Gigabit Ethernet Card (TPE-1000MPCI), the Penguin 10/100 USB Ethernet Network Adapter v1 (TPE-100NET1), and the Penguin 10/100 USB Ethernet Network Adapter v2 (TPE-100NET2)". This certification means that "products meet the FSF's standards in regard to users' freedom, control over the product, and privacy."

- The new PocketBeagle Linux computer is now available for $29.95 from Adafruit. According to Geeky Gadgets, the PocketBeagle "offers a powerful 1GHz AM3358 powered Linux single board computer with a tiny form factor and open source architecture". The article quotes Adafruit on the new SBC: "what differentiates the BeagleBone is that it has multiple I2C, SPI and UART peripherals (many boards only have one of each), built in hardware PWMs, analog inputs, and two separate 200MHz microcontroller systems called the PRU that can handle real-time tasks like displaying to RGB matrix displays or NeoPixels. It's not too much larger than our Feathers, but comes with 72 expansion pin headers, high-speed USB, 8 analog pins, 44 digital I/Os, and plenty of digital interface peripherals. You can also add a USB host connection by wiring a USB A socket to the broken out USB host connections labeled VI, D+, D-, ID and GND. Then plug in any USB Ethernet, Bluetooth, and Wi-Fi device with available Linux drivers."

- The official Raspberry Pi keyboard and mouse are now available. You can purchase them now from approved Raspberry Pi resellers. The keyboard is available in six layouts—English (UK), English (US), Spanish, French, German and Italian—with more in the works. The mouse is a " three-button, scroll-wheel optical device with Raspberry Pi logos on the base and cable, coloured to match the Pi case". View a video of the products for more details.

- SUSE is on track to become the largest independent Linux company. *ZDNet* reports that this is due to IBM acquiring Red Hat and SUSE's growth for the past seven straight years. The *ZDNet* post quotes SUSE CEO Nils Braukmann, "We believe that makes our status as a truly independent open-source company more important than ever. Our genuinely open-source solutions, flexible business practices, lack of enforced vendor lock-in, and exceptional service are more critical to customer and partner organizations, and our independence coincides with our single-minded focus on delivering what is best for them."

- Chef has announced it is releasing all of its software as open source. According to DevOps.com, "Chef has decided to open source its entire portfolio of IT automation software as part of an effort to make it easier for organizations to construct a DevOps pipeline using the company's software. A part of that effort, Chef also launched the Chef Enterprise Automation Stack—which combines Chef Infra for managing infrastructure, Chef InSpec for maintaining compliance, Chef Habitat for managing applications, Chef Automate for managing hybrid clouds and Chef Workstation, a starter kit for launching Chef—within a single distribution of Chef software. Chef Infra is the original Chef project around which the company was launched."

- Purism is partnering with Private Internet Access (PIA), "as its very first OEM partner to bring an unprecedented combination of tracking-free and encrypted tools and services to the people." From the Purism blog post: "By combining its signature VPN capabilities with Purism's leading secure hardware and software products, the two will create a first-of-its-kind bundle for users to set up a privacy protecting and secure environment out of the box. The addition of PIA

as a VPN partner strengthens Purism's growing roster of partners and services that make its Librem line the most comprehensive privacy and security focused offering on the market."

- RaspberryPi.org reports that you can now build a digital Etch-A-Sketch. The post notes that Martin Fitzpatrick built something called an "Etch-A-Snap", which is a Raspberry Pi Zero and camera module-connected Etch-A-Sketch: "Etch-A-Snap is (probably) the world's first Etch-A-Sketch Camera. Powered by a Raspberry Pi Zero (or Zero W), it snaps photos just like any other camera, but outputs them by drawing to an Pocket Etch-A-Sketch screen. Quite slowly." See Martin's Reddit post for more details.

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Digital Will, Part I: Requirements

Digital assets are becoming as important as physical assets, so how you do manage them after you die?

By Kyle Rankin

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks*, *DevOps Troubleshooting*, *The Official Ubuntu Server Book*, *Knoppix Hacks*, *Knoppix Pocket Reference*, *Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

When you lose a member of your family, you may find yourself at some point thinking about your own mortality, which then may lead you to think through preparations for your own death. I lost my father recently, but years before his death, he set up a will that described how to manage his estate, but he also made sure to share with me login details for his important financial accounts so I would have access when the time came. When the time did come to put his plans into practice, those details were invaluable.

All of this made me realize just how complicated it would be for someone to manage my own accounts in the event of my death, especially considering how much effort I've gone through to secure my computers and accounts. After all, unlike my dad, I don't use the same password for everything. What I realized I needed was the equivalent of a digital will: instructions and credentials so my next of kin

had everything they needed to access my accounts and manage my affairs. In this first article of what will be a two-part series, I describe the requirements and plans to create a digital will in a way that would be manageable for my next of kin while also not negatively affecting the security of my accounts. The second part of the article will describe how I implemented these plans.

Defining Terms

This digital will is based on many of the ideas behind a traditional will, and I intend on borrowing a lot of the framework and terms instead of "re-inventing the will". To get started, let me define a few terms, but I should make it clear that I'm not an attorney, so these are just loose definitions to describe how some common terms used in a will might be applied to this digital will:

- **Decedent/Testator:** the decedent is the person who has died, and the testator is the person who signs the will and whose will it is. For our purposes, this will be the same person—the person who currently controls these digital assets that will be transferred upon his or her death.

- **Beneficiary/Inheritor:** the person or persons who are receiving a gift of personal property from the decedent. For our purposes, this is the person or persons who will get control of digital accounts or other assets.

- **Administrator/Executor:** the person who is to oversee the administration of the estate and make sure the will is followed according to the testator's wishes. Often a testator will name a preferred executor in the will itself; other times, they are appointed by the court. In the case of a traditional will, the executor also may happen to be a beneficiary, but for some larger or more complicated estates it's often a third party selected for their financial or business know-how. For our purposes here, the executor will need technical know-how and will be the person who assists the beneficiary with getting access to accounts and managing any digital assets up to the point that the beneficiary can take over and will no longer need technical assistance.

Goals and Requirements

Unlike a traditional will, this digital will does *not* have a goal of defining who *gets* digital assets like online financial accounts—a traditional will already can define that sort of thing in an appropriate and legal way. Instead, the main goal of this digital will is to enable the executor to grant the beneficiary access to digital assets left behind. This main goal then helps with defining some related requirements.

Requirement 1: Simplicity

Dealing with a loved one's death and regular estate is difficult enough as it is. The extra complexity behind digital assets makes this even more difficult, and since you'll want to add security requirements on top, this very easily could result in a complicated and hard-to-follow process. Simplicity has to be a primary requirement, since you won't be around to help with the administration. This means however tempting it might be to use sophisticated cryptography algorithms or technologies, you need to make the digital will as foolproof and simple as possible.

Requirement 2: Documentation

Since I work in technology, I set up and maintain a number of systems at home. These include common household systems like wireless access points, a local file server and media center computers. Those systems are pretty common for people who are into computers, but as I have a sysadmin background, I also maintain email, web and DNS servers for domains that we own and that I and my family use for our main email and various blogs and other websites. All of these systems are largely undocumented, since I am the one who set them up and no one else maintains them, but obviously, that presents a problem if I die.

So one requirement for this project is to provide some kind of documentation for all of those systems. This documentation is not just about which systems exist (which is an important start), but it's also some level of detail on how to maintain those systems. The executor is the technical help here, but they can't be expected to perform their duties indefinitely. So the documentation also might need to cover how to migrate to a replacement system in the future, since some systems need more technical know-how to maintain long-term than the beneficiary may have. The person

writing the digital will is in the best place to make those determinations, as they know the skill levels and time commitment necessary to maintain the systems as well as the skill levels and free time available to both the executor and the beneficiary.

Requirement 3: Secure Transfer of Account Authentication

Whether you maintain local IT systems in your house or not, you still likely have a number of online accounts that you don't share with anyone else. Each of those accounts has its own set of credentials, and although some online services have support in place so that a beneficiary or next of kin can take over the decedent's account with a valid death certificate, many don't.

The digital will needs to provide a way to transfer access to those accounts over to the beneficiary, possibly with the help of the executor, without putting those accounts at risk from outside attackers. Ideally, the accounts would sit in a kind of digital trust so that the executor can't independently get access to those accounts prematurely—no matter how much you trust your executor, you probably don't want that person to have access to all your accounts and systems while you are alive. In some circumstances, you also might choose to prevent the beneficiary from getting premature access as well (for instance, if the beneficiary is your child). Even if you *do* trust your executor and beneficiary fully, you may not trust your full set of secrets on their systems since they could get hacked.

Requirement 4: Maintenance

If you ever have been responsible for technical documentation, you know how quickly that sort of thing can become out of date. This digital will process is no different, but it's even more important that it be kept up to date. This means you need to add an additional requirement that you build in some process to keep the documentation, credentials and everything else related to this digital will up to date via some periodic process.

Requirement 5: Fault Tolerance

Technology and people can fail, so this digital will needs to account for and be resilient to failure both in any technology it picks and mistakes or memory lapses

for any people involved. Systems should be redundant and account for failures and mistakes. You should be careful when choosing any solutions that rely on third parties that could go out of business or shut down a particular service they provide.

Requirement 6: FOSS

The technologies for this digital will should rely on free and open-source software (FOSS) not just because that matches my own ideals (and the ideals of *Linux Journal*), but also because a FOSS solution helps with the fault-tolerance requirement. FOSS software, even if it becomes unmaintained, still should be available for use in the future, whereas proprietary software or services may not.

Conclusion

Thinking through these requirements was hard, but not nearly as hard as figuring out an implementation that satisfies these requirements! In the next part of this article series, I will describe the solution I came up with for my own digital will. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

25TH ANNUAL

WITI

WOMEN IN TECHNOLOGY

SUMMIT

BECOME
CONNECTED
COACHED
INSPIRED

For 25 years running, the Women in Technology Summit has served as the premier gathering for more than 1,500 women and men from around the world, who converge in Silicon Valley for three special days to build relationships, collaborate, and update their technical and leadership skills.

JUNE 9-11, 2019 | SAN JOSE, CA

Women in Technology International, the leading advocate for innovation, inclusivity and STEAM, is celebrating 30 years of supporting and fostering women in technology.

WITI30
30TH ANNIVERSARY

REGISTER AT WITI.COM/SUMMIT

Introducing Mypy, an Experimental Optional Static Type Checker for Python

Reuven Lerner teaches Python, data science and Git to companies around the world. You can subscribe to his free, weekly "better developers" e-mail list, and learn from his books and courses at http://lerner.co.il. Reuven lives with his wife and children in Modi'in, Israel.

Tighten up your code and identify errors before they occur with `mypy`.

By Reuven M. Lerner

I've been using dynamic languages—Perl, Ruby and Python—for many years. I love the flexibility and expressiveness that such languages provide. For example, I can define a function that sums numbers:

```
def mysum(numbers):
    total = 0
    for one_number in numbers:
        total += one_number
    return total
```

The above function will work on any iterable that returns numbers. So I can run the above on a list, tuple or set of numbers. I can even run it on a dictionary whose keys are all numbers. Pretty great, right?

Yes, but for my students who are used to static, compiled languages, this is a very hard thing to get used to. After all, how can you make sure that no one passes you a string, or a number of strings? What if you get a list in which some, but not all, of the elements are numeric?

For a number of years, I used to dismiss such worries. After all, dynamic languages have been around for a long time, and they have done a good job. And really, if people are having these sorts of type mismatch errors, then maybe they should be paying closer attention. Plus, if you have enough testing, you'll probably be fine.

But as Python (and other dynamic languages) have been making inroads into large companies, I've become increasingly convinced that there's something to be said for type checking. In particular, the fact that many newcomers to Python are working on large projects, in which many parts need to interoperate, has made it clear to me that some sort of type checking can be useful.

How can you balance these needs? That is, how can you enjoy Python as a dynamically typed language, while simultaneously getting some added sense of static-typing stability?

One of the most popular answers is a system known as `mypy`, which takes advantage of Python 3's type annotations for its own purposes. Using `mypy` means that you can write and run Python in the normal way, gradually adding static type checking over time and checking it outside your program's execution.

In this article, I start exploring `mypy` and how you can use it to check for problems in your programs. I've been impressed by `mypy`, and I believe you're likely to see it deployed in a growing number of places, in no small part because it's optional, and thus allows developers to use it to whatever degree they deem necessary, tightening things up over time, as well.

Dynamic and Strong Typing

In Python, users enjoy not only dynamic typing, but also strong typing. "Dynamic" means that variables don't have types, but that values do. So you can say:

```
>>> x = 100
>>> print(type(x))
int

>>> x = 'abcd'
>>> print(type(x))
str

>>> x = [10, 20, 30]
>>> print(type(x))
list
```

As you can see, I can run the above code, and it'll work just fine. It's not particularly useful, per se, but it never would pass even a first-pass compilation in a statically compiled language. That's because in such languages, variables have types—meaning that if you try to assign an integer to a string variable, you'll get an error.

In a dynamic language, by contrast, variables don't have types at all. Running the `type` function, as I did above, doesn't actually return the variable's type, but rather the type of data to which the variable currently points.

Just because a language is dynamically typed doesn't mean that it's totally loosey-goosey, letting you do whatever you want. (And yes, that is the technical term.) For example, I can try this:

```
>>> x = 1
>>> y = '1'
>>> print(x+y)
```

That code will result in an error, because Python doesn't know how to add integers and strings together. It can add two integers (and get an integer result) or two strings (and get a string result), but not a combination of the two.

The `mysum` function that you saw earlier assigns 0 to the local "total" variable, and then adds each of the elements of `numbers` to it. This means that if `numbers` contains any non-numbers, you're going to be in trouble. Fortunately, `mypy` will be able to solve this problem for you.

Type Annotations

Python 3 introduced the idea of "type annotations," and as of Python 3.6, you can annotate variables, not just function parameters and return values. The idea is that you can put a colon (:) and then a type following parameter names. For example:

```
def hello(name:str):
    return f'Hello, {name}'
```

Here, I've given the `name` parameter a type annotation of `str`. If you've used a statically typed language, you might believe that this will add an element of type safety. That is, you might think that if I try to execute:

```
hello(5)
```

I will get an error. But in actuality, Python will ignore these type annotations completely. Moreover, you can use any object you want in an annotation; although it's typical to use a type, you actually can use anything.

This might strike you as completely ridiculous. Why introduce such annotations, if you're never going to use them? The basic idea is that coding tools and extensions will be able to use the annotations for their own purposes, including (as you'll see in just a bit) for the purposes of type checking.

This is important, so I'll repeat and stress it: type annotations are ignored by the Python

language, although it does store them in an attribute called `__annotations__`. For example, after defining the above `hello` function, you can look at its annotations, which are stored as a dictionary:

```
>>> hello.__annotations__
{'name': <class 'str'>}
```

Using Mypy

The `mypy` type checker can be downloaded and installed with the standard Python `pip` package installer. On my system, in a terminal window, I ran:

```
$ pip3 install –U mypy
```

The `pip3` reflects that I'm using Python 3, rather than Python 2. And the `–U` option indicates that I'd like to upgrade my installation of `mypy`, if the package has been updated since I last installed it on my computer. If you're installing this package globally and for all users, you might well need to run this as root, using `sudo`.

Once `mypy` is installed, you can run it, naming your file. For example, let's assume that hello.py looks like this:

```
def hello(name:str):
   return f"Hello, {name}"

print(hello('world'))
print(hello(5))
print(hello([10, 20, 30]))
```

If I run this program, it'll actually work fine. But I'd like to use that type annotation to ensure that I'm only invoking the function with a string argument. I can thus run, on the command line:

```
$ mypy ./hello.py
```

And I get the following output:

```
hello.py:7: error: Argument 1 to "hello" has incompatible type
 ↪"int"; expected "str"
hello.py:8: error: Argument 1 to "hello" has incompatible type
 ↪"List[int]"; expected "str"
```

Sure enough, `mypy` has identified two places in which the expectation that I've expressed with the type annotation—namely, that only strings will be passed as arguments to "hello"—has been violated. This doesn't bother Python, but it should bother you, either because the type annotation needs to be loosened up, or because (as in this case), it's calling the function with the wrong type of argument.

In other words, `mypy` won't tell you what to do or stop you from running your program. But it will try to give you warnings, and if you hook this together with a Git hook and/or with an integration and testing system, you'll have a better sense of where your program might be having problems.

Of course, `mypy` will check only where there are annotations. If you fail to annotate something, `mypy` won't be able to check it.

For example, I didn't annotate the function's return value. I can fix that, indicating that it returns a string, with:

```
def hello(name:str) -> str:
   return f"Hello, {name}"
```

Notice that Python introduced a new syntax (the `->` arrow), and allowed me to stick an annotation before the end-of-line colon, in order for annotations to work. The annotation dictionary has now expanded too:

```
>>> hello.__annotations__
{'name': <class 'str'>, 'return': <class 'str'>}
```

And in case you're wondering what Python will do if you have a local variable named `return` that conflicts with the return value's annotation…well, "return" is a reserved word and cannot be used as a parameter name.

More Sophisticated Checking

Let's go back to the `mysum` function. What will (and won't) `mypy` be able to check? For example, assume the following file:

```
def mysum(numbers:list) -> int:
    output = 0
    for one_number in numbers:
        output += one_number
    return output

print(mysum([10, 20, 30, 40, 50]))
print(mysum((10, 20, 30, 40, 50)))
print(mysum([10, 20, 'abc', 'def', 50]))
print(mysum('abcd'))
```

As you can see, I've annotated the `numbers` parameter to take only lists and to indicate that the function will always return integers. And sure enough, `mypy` catches the problems:

```
mysum.py:10: error:
    Argument 1 to "mysum" has incompatible type
           "Tuple[int, int, int, int, int]"; expected
             ↪"List[Any]"

mysum.py:12: error:
    Argument 1 to "mysum" has incompatible type
           "str"; expected "List[Any]"
```

The good news is that I've identified some problems. But in one case, I'm calling `mysum` with

a tuple of numbers, which should be fine, but is flagged as a problem. And in another case, I'm calling it with a list of both integers and strings, but that's seen as just fine.

I'm going to need to tell `mypy` that I'm willing to accept not just a list, but any sequence, such as a tuple. Fortunately, Python now has a `typing` module that provides you with objects designed for use in such circumstances. For example, I can say:

```
from typing import Sequence

def mysum(numbers:Sequence) -> int:
    output = 0
    for one_number in numbers:
        output += one_number
    return output
```

I've grabbed `Sequence` from the `typing` module, which includes all three Python sequence types—strings, lists and tuples. Once I do that, all of the `mypy` problems disappear, because all of the arguments are sequences.

That went a bit overboard, admittedly. What I really want to say is that I'll accept any sequence whose elements are integers. I can state that by changing my function's annotations to be:

```
from typing import Sequence

def mysum(numbers:Sequence[int]) -> int:
    output = 0
    for one_number in numbers:
        output += one_number
    return output
```

Notice that I've modified the annotation to be `Sequence[int]`. In the wake of that change, `mypy` has now found lots of problems:

```
mysum.py:13: error: List item 2 has incompatible type "str";
 ↪expected "int"
mysum.py:13: error: List item 3 has incompatible type "str";
 ↪expected "int"
mysum.py:14: error: Argument 1 to "mysum" has incompatible type
 ↪"str"; expected "Sequence[int]"
```

I'd call this a big success. If someone now tries to use my function with the wrong type of value, it'll call them out on it.

But wait: do I really only want to allow for lists and tuples? What about sets, which also are iterable and can contain integers? And besides, what's this obsession with integers—shouldn't I also allow for floats?

I can solve the first problem by saying that I'll take not a `Sequence[int]`, but `Iterable[int]`—meaning, anything that is iterable and returns integers. In other words, I can say:

```
from typing import Iterable

def mysum(numbers:Iterable[int]) -> int:
    output = 0
    for one_number in numbers:
        output += one_number
    return output
```

Finally, how can I allow for either integers or strings? I use the special `Union` type, which lets you combine types together in square brackets:

```
from typing import Iterable, Union

def mysum(numbers:Iterable[Union[int, float]]) ->
 ↪Union[int,float]:
```

```
    output = 0
    for one_number in numbers:
        output += one_number
    return output
```

But if I run `mypy` against this code, and try to call `mysum` with an iterable containing at least one float, I'll get an error:

```
mysum.py:9: error: Incompatible types in assignment
 ↪(expression has type "float", variable has type "int")
```

What's the problem? Simply put, when I create `output` as a variable, I'm giving it an integer value. And then, when I try to add a floating-point value to it, I get a warning from `mypy`. So, I can silence that by annotating the variable:

```
def mysum(numbers:Iterable[Union[int, float]])
 ↪-> Union[int,float]:
    output : Union[int,float] = 0
    for one_number in numbers:
        output += one_number
    return output
```

Sure enough, the function is now pretty well annotated. I'm too experienced to know that this will catch and solve all problems, but if others on my team, who want to use my function, use `mypy` to check the types, they'll get warnings. And that's the whole point here, to catch problems before they're even close to production. ■

Resources

You can read more about `mypy` here. That site has documentation, tutorials and even information for people using Python 2 who want to introduce `mypy` via comments (rather than annotations).

Breaking Up Apache Log Files for Analysis

Dave tackles analysis of the ugly Apache web server log.

By Dave Taylor

Dave Taylor has been hacking shell scripts on Unix and Linux systems for a really long time. He's the author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*. You can find him on Twitter as @DaveTaylor, and you can reach him through his tech Q&A site Ask Dave Taylor.

I know, in my last article I promised I'd jump back into the mail merge program I started building a while back. Since I'm having some hiccups with my AskDaveTaylor.com web server, however, I'm going to claim editorial privilege and bump that yet again.

What I need to do is be able to process Apache log files and isolate specific problems and glitches that are being encountered—a perfect use for a shell script. In fact, I have a script of this nature that offers basic analytics in my book *Wicked Cool Shell Scripts* from O'Reilly, but this is a bit more specific.

Oh Those Ugly Log Files

To start, let's take a glance at a few lines out of the latest log file for the site:

```
$ head sslaccesslog_askdavetaylor.com_3_8_2019
```

```
18.144.59.52 - - [08/Mar/2019:06:10:09 -0600] "GET /wp-content/
↪themes/jumpstart/framework/assets/js/nivo.min.js?ver=3.2
 ↪HTTP/1.1" 200 3074
"https://www.askdavetaylor.com/how-to-play-dvd-free-windows-
↪10-win10/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
 ↪AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
 ↪64.0.3282.140 Safari/537.36 Edge/18.17763 X-Middleton/1"
 ↪52.53.151.37 - - [08/Mar/2019:06:10:09 -0600] "GET
 ↪/wp-includes/js/jquery/jquery.js?ver=1.12.4 HTTP/1.1"
 ↪200 33766 "https://www.askdavetaylor.com/how-to-play
↪-dvd-free-windows-10-win10/" "Mozilla/5.0 (Windows NT
 ↪10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
 ↪Chrome/64.0.3282.140 Safari/537.36 Edge/18.17763
 ↪X-Middleton/1" 18.144.59.52 - - [08/Mar/2019:06:10:09
 ↪-0600] "GET /wp-content/plugins/google-analytics-for-
↪wordpress/assets/js/frontend.min.js?ver=7.4.2 HTTP/1.1"
 ↪200 2544 "https://www.askdavetaylor.com/how-to-play
↪-dvd-free-windows-10-win10/"
 ↪"Mozilla/5.0 (Windows NT 10.0; Win64; x64)
 ↪AppleWebKit/537.36 (KHTML, like Gecko)
 ↪Chrome/64.0.3282.140 Safari/537.36 Edge/18.17763
 ↪X-Middleton/1"
```

It's big and ugly, right? Okay, then let's just isolate a single entry to see how it's structured:

```
18.144.59.52 - - [08/Mar/2019:06:10:09 -0600] "GET
 ↪/wp-content/themes/jumpstart/framework/assets/js/
↪nivo.min.js?ver=3.2 HTTP/1.1" 200 3074
"https://www.askdavetaylor.com/how-to-play-dvd-free-windows-
↪10-win10/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140
 ↪Safari/537.36 Edge/18.17763 X-Middleton/1"
```

That's still obfuscated enough to kick off a migraine!

Fortunately, the Apache website has a somewhat clearer explanation of what's known as the custom log file format that's in use on my server. Of course, it's described in a way that only a programmer could love:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
 ↪'\"%{User-agent}i\""
```

That's enough info to help decode the log entry. I'll define each of the percent-format sequences as I go, so you can get a sense of how it's all tied together too:

```
%h = IP Address = 18.144.59.52
%l = ID of client = -
%u = UserID of client = -
%t = Time of request = [08/Mar/2019:06:10:09 -0600]
%r = Request = "GET /wp-content/themes/jumpstart/framework/
↪assets/js/nivo.min.js?ver=3.2 HTTP/1.1"
%>s = Status code = 200
%b = Size of request = 3074
Referrer = "https://www.askdavetaylor.com/how-to-play-dvd-
↪free-windows-10-win10/"
User Agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
 ↪AppleWebKit/537.36 (KHTML, like Gecko)
 ↪Chrome/64.0.3282.140 Safari/537.36 Edge/18.17763
 ↪X-Middleton/1"
```

Or, to make it a bit clearer yet:

```
IP - - TIMESTAMP REQUEST STATUS SIZE REFERRER USERAGENT
```

This becomes complicated to parse because there are two different types of field separator: a space for each of the major fields, but since some of the values can

contain spaces, quotes to delimit the start/end of fields Request, Referrer and User Agent.

As a general rule, shell utilities aren't so good at these sort of mixed field separators, so it's time for a bit of out-of-the-box thinking!

Breaking Down Fields with Dissimilar Delimiters

It's true that the fields are divided up with dissimilar delimiters (say that ten times fast), but you can process the information in stages. You can examine the line by just processing the quote delimiter with this clumsy code block:

```
while read logentry
do
   echo "f1 = $(echo "$logentry" | cut -d\" -f1)"
   echo "f2 = $(echo "$logentry" | cut -d\" -f2)"
   echo "f3 = $(echo "$logentry" | cut -d\" -f3)"
   echo "f4 = $(echo "$logentry" | cut -d\" -f4)"
   echo "f5 = $(echo "$logentry" | cut -d\" -f5)"
   echo "f6 = $(echo "$logentry" | cut -d\" -f6)"
done < $accesslog
```

Since it's just an interim step on the development of the final shell script, I'm not going to bother cleaning it up or making it more efficient.

Running this against the first line of the `accesslog`, here's what's revealed:

```
f1 = 18.144.59.52 - - [08/Mar/2019:06:10:09 -0600]
f2 = GET /wp-content/themes/jumpstart/framework/assets/
↪js/nivo.min.js?ver=3.2 HTTP/1.1
f3 =  200 3074
f4 = https://www.askdavetaylor.com/how-to-play-dvd-free-
↪windows-10-win10/
f5 =
```

```
f6 = Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140
Safari/537.36 Edge/18.17763 X-Middleton/1
```

What's important to notice here is field 3. Field 3 (`f3`) has both the return code—`200`, in this case—and the total number of bytes in this transaction, `3074`.

This means that if `f3` is then divided by the space delimiter, you can identify both return code and bytes easily enough:

```
f3=$(echo "$logentry" | cut -d\" -f3)
   returncode="$(echo $f3 | cut -f1 -d\  )"
   bytes="$(echo $f3 | cut -f2 -d\  )"
```

Using a space as a delimiter makes for a weird-looking command line, as you can see, but the \ forces the very next character to be interpreted as the specified value, first a double quote, then a space character.

Extracting Just the Errors

Now, can you spin through the entire log file and just pull out error codes? Sure you can, with just a simplification and tweak of the `while` loop:

```
while read logentry
do
   f3=$(echo "$logentry" | cut -d\" -f3)
      returncode="$(echo $f3 | cut -f1 -d\  )"
      bytes="$(echo $f3 | cut -f2 -d\  )"

   echo "$entry: returncode = $returncode, bytes = $bytes"
   entry=$(( $entry + 1 ))
done < $accesslog
```

Since a return code of `200` is a success, it's easy to `grep -v` and see what other

return codes show up in the log file:

```
$ sh breakdown.sh  | grep -v 200
113: returncode = 405, bytes = 42
138: returncode = 405, bytes = 42
177: returncode = 301, bytes = -
183: returncode = 301, bytes = -
186: returncode = 405, bytes = 42
187: returncode = 404, bytes = 11787
220: returncode = 404, bytes = 11795
279: returncode = 405, bytes = 42
397: returncode = 301, bytes = -
```

Error 405 is (according to the W3 Web standards site) "Method Not Allowed", while 301 is "Moved Permanently", and 404 is a standard "Not Found" error when someone requests a resource that the server cannot find.

Useful, but let's take the next step. For every query where the return code is not a `200` "OK" status, let's show the original log file line in question. This time, let's modify the script to do the `200` filtering too:

```
while read logentry
do
   f3=$(echo "$logentry" | cut -d\" -f3)
      returncode="$(echo $f3 | cut -f1 -d\  )"
      bytes="$(echo $f3 | cut -f2 -d\  )"

   if [ $returncode != "200" ] ; then
     echo "$returncode ($entry): $logentry"
   fi

   entry=$(( $entry + 1 ))
done < $accesslog
```

The results then look like this:

```
$ sh breakdown.sh
405 (113): 3.122.179.106 - - [08/Mar/2019:06:10:11 -0600]
"GET /xmlrpc.php HTTP/1.1" 405 42 "-" "Mozilla/5.0 (X11;
Linux i686; rv:2.0.1) Gecko/20100101 Firefox/4.0.1
 ↪X-Middleton/1"
405 (138): 35.180.37.73 - - [08/Mar/2019:06:10:21 -0600]
"GET /xmlrpc.php HTTP/1.1" 405 42 "-" "Mozilla/5.0 (X11;
Linux i686; rv:2.0.1) Gecko/20100101 Firefox/4.0.1
 ↪X-Middleton/1"
301 (177): 34.239.180.102 - - [08/Mar/2019:06:10:30 -0600]
"GET /how_do_i_restructure_my_wordpress_blog_without_losing_seo
 ↪HTTP/1.1" 301 - "-" "Mozilla/5.0 (Windows NT 6.1;
 ↪WOW64; rv:29.0) Gecko/20120101 Firefox/29.0
 ↪X-Middleton/1"
```

It's useful to be able to see the log file entry line, the return error code and the full log file entry line. Is there a pattern? Do they all have the same user agent (for example, a bot)? Are they from the same IP address? A pattern based on time of day?

With a judicious use of `wc`, I also can ascertain that this particular log file encompasses 99,309 total hits, of which 4,393 (4.4%) are non-200 results.

Another useful feature for this script would be to create multiple output files automatically, one per error code. I shall leave that, however, as an exercise for you, dear reader!

And, for my next article, I'll jump back into that mail merge script! ■

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Decentralized Certificate Authority and Naming

Free and open source contributors only:

handshake.org/signup

What's New in Kernel Development

Zack Brown is a tech journalist at *Linux Journal* and *Linux Magazine*, and is a former author of the "Kernel Traffic" weekly newsletter and the "Learn Plover" stenographic typing tutorials. He first installed Slackware Linux in 1993 on his 386 with 8 megs of RAM and had his mind permanently blown by the Open Source community. He is the inventor of the *Crumble* pure strategy board game, which you can make yourself with a few pieces of cardboard. He also enjoys writing fiction, attempting animation, reforming Labanotation, designing and sewing his own clothes, learning French and spending time with friends'n'family.

By Zack Brown

KUnit and Assertions

KUnit has been seeing a lot of use and development recently. It's the kernel's new unit test system, introduced late last year by **Brendan Higgins**. Its goal is to enable maintainers and other developers to test discrete portions of kernel code in a reliable and reproducible way. This is distinct from various forms of testing that rely on the behavior of the system as a whole and, thus, do not necessarily always produce identical results.

Lately, Brendan has submitted patches to make KUnit work conveniently with "assertions". Assertions are like conditionals, but they're used in situations where only one possible condition should be true. It shouldn't be possible for an assertion to be false. And so if it is, the assertion triggers some kind of handler that the developer then uses to help debug the reasons behind the failure.

Unit tests and assertions are to some extent in opposition to each other—a unit test could trigger an assertion when the intention was to exercise the code being tested. Likewise, if a unit test does trigger an assertion, it could mean that the underlying assumptions made by the unit test can't be relied on,

and so the test itself may not be valid.

In light of this, Brendan submitted code for KUnit to be able to break out of a given test, if it triggered an assertion. The idea behind this was that the assertion rendered the test invalid, and KUnit should waste no time, but proceed to the next test in the queue.

There was nothing particularly controversial in this plan. The controversial part came when **Frank Rowand** noticed that Brendan had included a call to `BUG()`, in the event that the unit test failed to abort when instructed to do so. That particular situation never should happen, so Brendan figured it didn't make much difference whether there was a call to `BUG()` in there or not.

But Frank said, "You will just annoy Linus if you submit this." He pointed out that the `BUG()` was a means to produce a kernel panic and hang the entire system. In Linux, this was virtually never an acceptable solution to any problem.

At first, Brendan just shrugged, since as he saw it, KUnit was part of the kernel's testing infrastructure and, thus, never would be used on a production system. It was strictly for developers only. And in that case, he reasoned, what difference would it make to have a `BUG()` here and there between friends? Not to mention the fact that, as he put it, the condition producing the call to `BUG()` never should arise.

But, Frank said this wasn't good enough. He said that whether you felt that KUnit belonged or didn't belong in production systems, it almost certainly would find its way into production systems in the real world. That's just how these things go. People do what isn't recommended. But even if that were not the case, said Frank, non-production systems likewise should avoid calling `BUG()`, unless crashing the system were the only way to avoid actual data corruption.

Brendan had no serious objection to ditching the call to `BUG()`, he was just posing questions, because it seemed odd that there would be any problem. But, he was fine with ditching it.

So the feature remains, while the error handling will change. An interesting thing about this particular debate is that it underscores the variety of conflicts that can emerge with so many debugging and error-handling aspects of the kernel. All sorts of conflicts and race conditions might emerge.

For example, a developer might write a new driver and want to test how it behaves under heavy load. So they'll run a memory-intensive process while using their driver, only to discover that the kernel's out-of-memory (OOM) killer kills the process generating the load, before the key test situation can be triggered within the driver.

It's amazing to consider the sheer quantity of testing and debugging features that have encrusted themselves on every aspect of the Linux kernel development process. Even **git** itself, the revision control system created by **Linus Torvalds** specifically to host kernel development, is itself a debugging tool that ensures it is possible to identify and possibly revert changes that turn out to cause a problem. In addition to everything else, there also are a wide array of automated systems running within a variety of private enterprises. Some of those load up running systems with particular workloads; some read the source code directly, looking for patterns. It's impossible to know the full variety and extent of testing that the Linux kernel receives on a daily basis.

Crazy Compiler Optimizations

Kernel development is always strange. **Andrea Parri** recently posted a patch to change the order of memory reads during multithreaded operation, such that if one read depended upon the next, the second could not actually occur before the first.

The problem with this was that the bug never could actually occur, and the fix made the kernel's behavior less intuitive for developers. **Peter Zijlstra**, in particular, voted nay to this patch, saying it was impossible to construct a physical system capable of triggering the bug in question.

And although Andrea agreed with this, he still felt the bug was worth fixing, if only for its theoretical value. Andrea figured, a bug is a bug is a bug, and they should be fixed. But Peter objected to having the kernel do extra work to handle conditions that could

never arise. He said, “what I do object to is a model that’s weaker than any possible sane hardware.”

Will Deacon sided with Peter on this point, saying that the underlying hardware behaved a certain way, and the kernel’s current behavior mirrored that way. He remarked, “the majority of developers are writing code with the underlying hardware in mind and so allowing behaviours in the memory model which are counter to how a real machine operates is likely to make things more confusing, rather than simplifying them!”

Still, there were some developers who supported Andrea’s patch. **Alan Stern**, in particular, felt that it made sense to fix bugs when they were found, but that it also made sense to include a comment in the code, explaining the default behavior and the rationale behind the fix, even while acknowledging the bug never could be triggered.

But, Andrea wasn’t interested in forcing his patch through the outstretched hands of objecting developers. He was happy enough to back down, having made his point.

It was actually **Paul McKenney**, who had initially favored Andrea’s patch and had considered sending it up to Linus Torvalds for inclusion in the kernel, who identified some of the deeper and more disturbing issues surrounding this whole debate. Apparently, it cuts to the core of the way kernel code is actually compiled into machine language. Paul said:

> We had some debates about this sort of thing at the C++ Standards Committee meeting last week.
>
> Pointer provenance and concurrent algorithms, though for once not affecting RCU! We might actually be on the road to a fix that preserves the relevant optimizations while still allowing most (if not all) existing concurrent C/C++ code to continue working correctly. (The current thought is that loads and stores involving inline assembly, C/C++ atomics, or volatile get their provenance stripped. There may need to be some other mechanisms for plain C-language loads and stores in some cases as well.)

> But if you know of any code in the Linux kernel that needs to compare pointers, one of which might be in the process of being freed, please do point me at it. I thought that the smp_call_function() code fit, but it avoids the problem because only the sending CPU is allowed to push onto the stack of pending smp_call_function() invocations.
>
> That same concurrent linked stack pattern using cmpxchg() to atomically push and xchg() to atomically pop the full list -would- have this problem. The old pointer passed to cmpxchg() might reference an object that was freed between the time that the old pointer was loaded and the time that the cmpxchg() executed. One way to avoid this is to do the push operation in an RCU read-side critical section and use kfree_rcu() instead of kfree(). Of course, code in the idle loop or that might run on offline CPUs cannot use RCU, plus some data structures are not happy with kfree_rcu() delays, so...

In other words, the issue of how the C compiler should treat pointers depends to some extent on whether they are pointers at all. There's nothing about a pointer that distinguishes it from any other number, except that the compiler knows it's a pointer and can therefore do certain things with it that wouldn't make sense in other contexts. It's this issue of the origins of a number—that is, their provenance—that the standards committee was trying to resolve. The reason any of this is useful and relevant is that the compiler can only optimize code to be faster and more efficient if it can understand what's happening and what's going to happen.

Peter poked around online until he found a paper describing the situation in detail.

It horrified him. His conclusion was, "that's all massive bong-hits. That's utterly insane. Even the proposed semantics are crazy."

Paul did not dissent from that view, though obviously more efficient code is better than less efficient code, and the compiler should go to whatever extremes it can manage to achieve it.

Paul said that none of this was new. In fact, it all dated back 20 years and more to the

relatively early days of multithreaded operation. There were, Paul said, a variety of approaches, and he said he hoped to be able to show the kernel folks some of what the **GCC** folks were thinking on the matter to get feedback and suggestions.

Peter still was a bit freaked out by the situation. In particular, he was concerned about whether the compiler could produce reliable code at all. He remarked, "at the very least we should get this fixed and compile a kernel with the fixed compiler to see what (if anything) changes in the generated code and analyze the changes (if any) to make sure we were ok (or not)."

The GNU C compiler is definitely filled with insanity. The whole question of how to convert C code into the best possible machine code is one that can never fully be answered—and in fact, the question continually changes as new CPUs come out on the market. Not to mention that the compiler also has to work around processor-specific security flaws like the ones plaguing **Intel** chips in recent years.

Add to this the fact that GCC needs to produce good code not just for the Linux kernel, but for any coding project that someone might dream up. So GCC has to remain both highly specialized and highly generalized at the same time. It makes perfect sense that its dark innards would be dark and innardly.

CGroup Interactions

CGroups are under constant development, partly because they form the core of many commercial services these days. An amazing thing about this is that they remain an unfinished project. Isolating and apportioning system elements is an ongoing effort, with many pieces still to do. And because of security concerns, it never may be possible to present a virtual system as a fully independent system. There always may be compromises that have to be made.

Recently, **Andrey Ryabinin** tried to fix what he felt was a problem with how CGroups dealt with low-memory situations. In the current kernel, low-memory situations would cause Linux to recuperate memory from all CGroups equally. But instead of being fair, this would penalize any CGroup that used memory efficiently and reward those

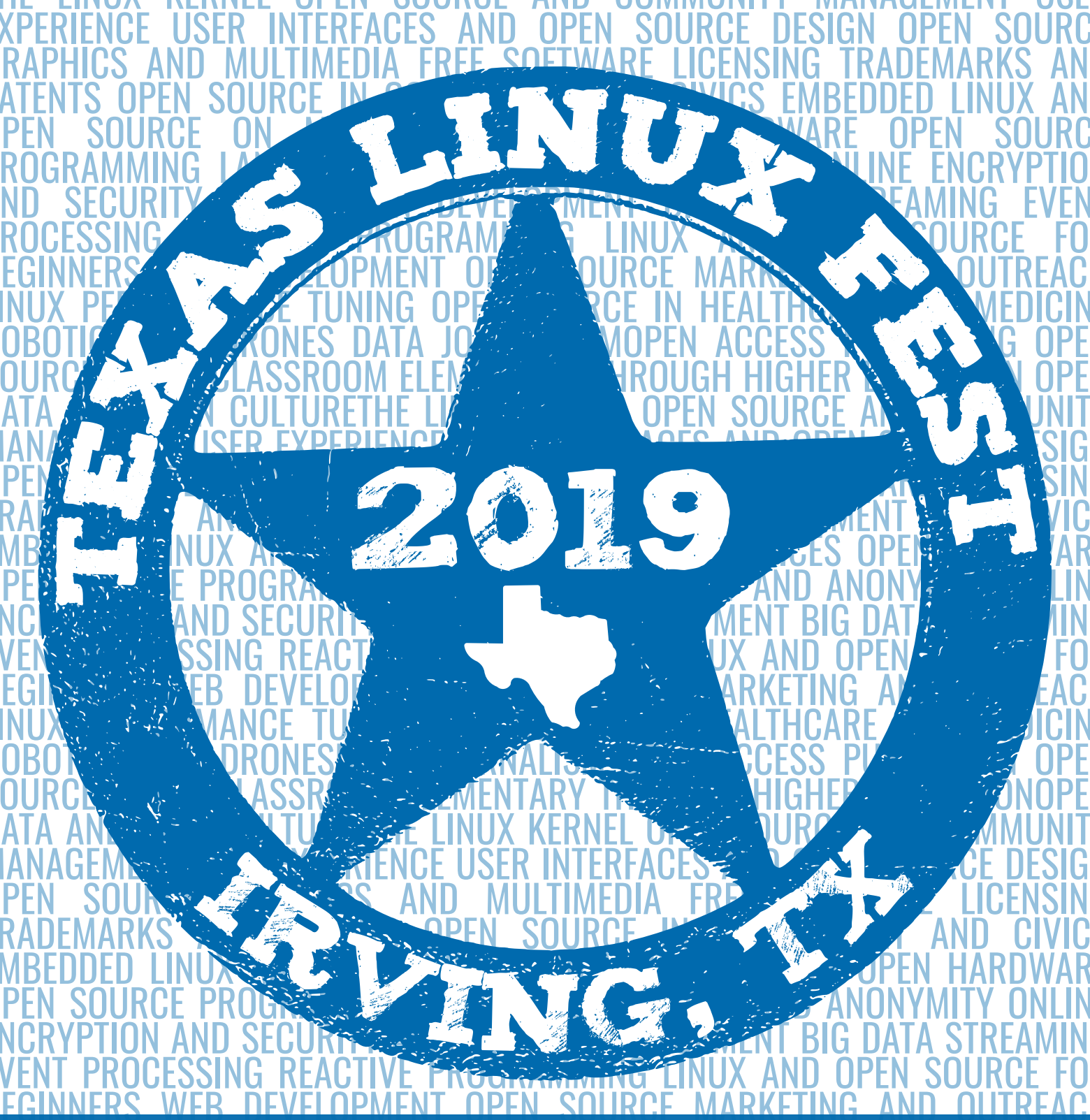
CGroups that allocated more memory than they needed.

Andrey's solution to this was to have Linux recuperate unused memory from CGroups that had it, before recuperating any from those that were in heavy use. This would seem to be even less fair than the original behavior, because only certain CGroups would be targeted and not others.

Andrey's idea garnered support from folks like **Rik van Riel**. But not everyone was so enthralled. **Roman Gushchin**, for example, pointed out that the distinction between active and unused memory was not as clear as Andrey made it out to be. The two of them debated this issue quite a bit, because the whole issue of fair treatment hangs in the balance. If Andrey's whole point is to prevent CGroups from "gaming the system" to ensure more memory for themselves, then the proper approach to low-memory conditions depends on being able to identify clearly which CGroups should be targeted for reclamation and which should be left alone.

At the same time, the situation could be seen as a security concern, with an absolute need to protect independent CGroups from each other. If so, something like Andrey's patch would be necessary, and many more security-minded developers would start to take an interest in getting the precise details exactly right.

Note: if you're mentioned in this article and want to send a response, please send a message with your response text to ljeditor@linuxjournal.com *and we'll run it in the next Letters section and post it on the website as an addendum to the original article.* ■



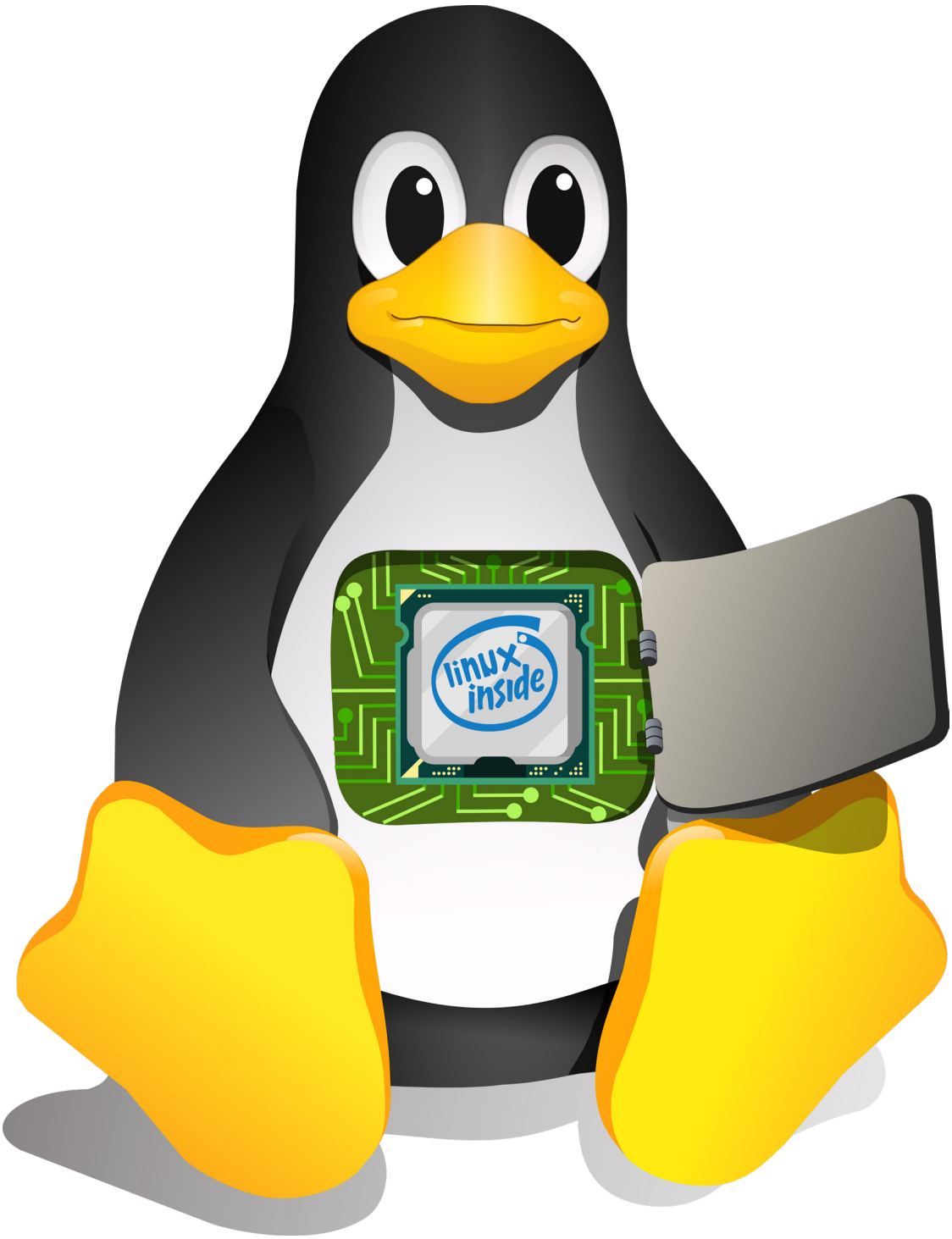
May 31st - June 1st

Irving Convention Center

texaslinuxfest.org

DEEP DIVE

THE KERNEL



What Does It Take to Make a Kernel?

The kernel this. The kernel that. People often refer to one operating system's kernel or another without truly knowing what it does or how it works or what it takes to make one. What does it take to write a custom (and non-Linux) kernel?

By Petros Koutoupis

So, what am I going to do here? In June 2018, I wrote a **guide to build a complete Linux distribution from source packages**, and in January 2019, I **expanded on that guide** by adding more packages to the original guide. Now it's time to dive deeper into the custom operating system topic. This article describes how to write your very own kernel from scratch and then boot up into it. Sounds pretty straightforward, right? Now, don't get *too* excited here. This kernel won't do much of anything. It'll print a few messages onto the screen and then halt the CPU. Sure, you can build on top of it and create something more, but that is not the purpose of this article. My main goal is to provide you, the reader, with a deep understanding of how a kernel is written.

Once upon a time, in an era long ago, embedded Linux was not really a *thing*. I know that sounds a bit crazy, but it's true! If you worked with a microcontroller, you were given (from the vendor) a specification, a design sheet, a manual of all its registers and nothing more. Translation: *you had to write your own operating system (kernel included) from scratch*. Although this guide assumes the standard generic 32-bit x86 architecture, a lot of it reflects what had to be done back in the day.

The exercises below require that you install a few packages in your preferred Linux distribution. For instance, on an Ubuntu machine, you will need the following:

- binutils
- gcc
- grub-common
- make
- nasm
- xorriso

An Extreme Crash Course into the Assembly Language

Note: I'm going to simplify things by pretending to work with a not-so-complex 8-bit microprocessor. This doesn't reflect the modern (and possibly past) designs of any commercial processor.

When the designers of a microprocessor create a new chip, they will write some very specialized microcode for it. That microcode will contain defined operations that are accessed via operation codes or *opcodes*. These defined opcodes contain instructions (for the microprocessor) to add, subtract, move values and addresses and more. The processor will read those opcodes as part of a larger command format. This format will consist of fields that hold a series of binary numbers—that is, 0s and 1s. Remember, this processor understands only high (the 1s) and low (the 0s) signals, and when those signals (as part of an instruction) are fed to it in the proper sequence, the processor will parse/interpret the instruction and then execute it.

Now, what exactly is assembly language? It's as close to machine code as you can get when programming a microprocessor. It is human-readable code based on the machine's supported instruction set and not just a series of binary numbers. I guess you *could* memorize all the binary numbers (in their proper sequence) for every instruction, but it wouldn't make much sense, especially if you can simplify code writing with more human-readable commands.

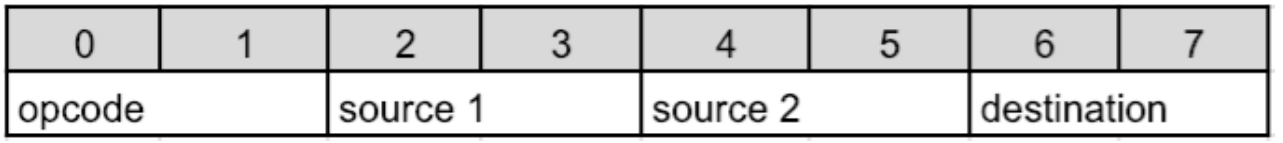


Figure 1. A Command Structure for the Made-Up Processor

This make-believe and completely unrealistic processor supports only four instructions of which the ADD instruction maps to an opcode of 00 in binary code, and SUB (or subtract) maps to an opcode of 01 in binary. You'll be accessing four total CPU memory registers: A or 00, B or 01, C or 10 and D or 11.

Using the above command structure, your compiled code will send the following instruction:

```
ADD A, B, C
```

Or, "add the contents of A and B and store them into register C" in the following binary machine language format:

```
00000110
```

Let's say you want to subtract A from C and store it in the B register. The human-readable code would look like the following:

```
SUB C, A, D
```

And, it will translate to the following machine code for the processor's microcode to process:

```
01100011
```

As you would expect, the more advanced the chip (16-bit, 32-bit, 64-bit), the more

instructions and larger address spaces are supported.

The Boot Code

The assembler I'm using in this tutorial is called NASM. The open-source NASM, or the Net-Wide Assembler, will *assemble* the assembly code into a file format called object code. The object file generated is an intermediate step to produce the executable binary or program. The reason for this intermediate step is that a single large source code file may end up being cut up into smaller source code files to make them more manageable in both size and complexity. For instance, when you compile the C code, you'll instruct the C compiler to produce only an object file. All object code (created from your ASM and C files) will form bits and pieces of your kernel. To finalize the compilation, you'll use a *linker* to take all necessary object files, combine them, and then produce the program.

The following code should be written to and saved in a file named boot.asm. You should store the file in the dedicated working directory for the project.

boot.asm

```
bits 32

section .multiboot                ;according to multiboot spec
        dd 0x1BADB002             ;set magic number for
                                  ;bootloader
        dd 0x0                    ;set flags
        dd - (0x1BADB002 + 0x0)   ;set checksum

section .text
global start
extern main                       ;defined in the C file

start:
        cli                       ;block interrupts
```

```
        mov esp, stack_space     ;set stack pointer
        call main
        hlt                      ;halt the CPU

section .bss
resb 8192                        ;8KB for stack
stack_space:
```

So, this looks like a bunch of nonsensical gibberish, right? It isn't. Again, this is supposed to be human-readable code. For instance, under the `multiboot` section, and in the proper order of the multiboot specification (refer to the section labeled "References" below), you're defining three *double words* variables. Wait, what? What is a double word? Well, let's take a step back. The assembly DD pseudo-instruction translates to Define Double (word), which on an x86 32-bit system is 4 bytes (32-bits). A DW or Define Word is 2 bytes (or 16 bits), and moving even further backward, a DB or Define Byte is 8-bits. Think of it as your *integers*, *short* and *long* in your high-level coding languages.

Note: pseudo-instructions are not real x86 machine instruction. They are special instructions supported by the assembler and for the assembler to help facilitate memory initialization and space reservation.

Below the `multiboot` section, you have a section labeled `text`, which is shortly followed by a function labeled `start`. This `start` function will set up the environment for your main kernel code and then execute that kernel code. It starts with a `cli`. The CLI command, or Clear Interrupts Flag, clears the `IF` flag in the `EFLAGS` register. The following line moves the empty `stack_space` function into the Stack Pointer. The Stack Pointer is small register on the microprocessor that contains the address of your program's last request from a Last-In-First-Out (LIFO) data buffer referred to as a Stack. The example assembly program will call the `main` function defined in your C file (see below) and then halt the CPU. If you look above, this is telling the assembler via the `extern main` line that the code for this function exists outside this file.

The Kernel's Main Function

So, you wrote your boot code, and your boot code knows that there is an external `main` function it needs to load into, but you don't have an external `main` function—at least, not yet. Create a file in the same working directory, and name it kernel.c. The file's contents should be the following:

kernel.c

```
#define VGA_ADDRESS 0xB8000   /* video memory begins here. */

/* VGA provides support for 16 colors */
#define BLACK 0
#define GREEN 2
#define RED 4
#define YELLOW 14
#define WHITE_COLOR 15

unsigned short *terminal_buffer;
unsigned int vga_index;

void clear_screen(void)
{
    int index = 0;
    /* there are 25 lines each of 80 columns;
       each element takes 2 bytes */
    while (index < 80 * 25 * 2) {
            terminal_buffer[index] = ' ';
            index += 2;
    }
}

void print_string(char *str, unsigned char color)
{
```

```
    int index = 0;
    while (str[index]) {
            terminal_buffer[vga_index] = (unsigned
             ↪lsshort)str[index]|(unsigned short)color << 8;
            index++;
            vga_index++;
    }
}

void main(void)
{
    /* TODO: Add random f-word here */
    terminal_buffer = (unsigned short *)VGA_ADDRESS;
    vga_index = 0;

    clear_screen();
    print_string("Hello from Linux Journal!", YELLOW);
    vga_index = 80;    /* next line */
    print_string("Goodbye from Linux Journal!", RED);
    return;
}
```

If you scroll all the way to the bottom of the C file and look inside the `main` function, you'll notice it does the following:

- Assigns the start address of your video memory to the string buffer.
- Resets your internal location marker for where you are in that string buffer.
- Clears the terminal screen.
- Prints a message (in one color).

- Sets your internal location marker for the next line.
- Prints another message (in another color).
- And, returns back to the boot code (where, if you recall, it halts the CPU).

In the current x86 architecture, your video memory is running in protected mode and starts at memory address `0xB8000`. So, everything video-related will start from this address space and will support up to 25 lines with 80 ASCII characters per line. Also, the video mode in which this is running supports up to 16 colors (of which I added a few to play with at the top of the C file).

Following these video definitions, a global array is defined to map to the video memory and an index to know where you are in that video memory. For instance, the index starts at 0, and if you want to move to the first character space of the next line on the screen, you'll need to increase that index to 80, and so on.

As the names of the following two functions imply, the first clears the entire screen with an ASCII empty character, and the second writes whatever string you pass into it. *Note that the expected input for the video memory buffer is 2 bytes per character. The first of the two is the character you want to output, while the second is the color. This is made more obvious in the* `print_string()` *function, where the color code is actually passed into the function.*

Anyway, following those two functions is the `main` routine with its actions already mentioned above. Remember, this is a learning exercise, and this kernel will not do anything special other than print a few things to the screen. And aside from adding real functions, this kernel code is definitely missing some profanity. (You can add that later.)

In the real world...

Every kernel will have a `main()` routine (spawned by a bootloader), and within that

main routine, all the proper system initialization will take place. In a real and functional kernel, the main routine eventually will drop into an infinite `while()` loop where all future kernel functions take place or spawn a thread accomplishing pretty much the same thing. Linux does this as well. The bootloader will call the `start_kernel()` routine found in init/main.c, and in turn, that routine will spawn an init thread.

Linking It All Together

As mentioned previously, the linker serves a very important purpose. It is what will take all of the random object files, put them together and provide a bootable single binary file (your kernel).

linker.ld

```
OUTPUT_FORMAT(elf32-i386)
ENTRY(start)
SECTIONS
 {
   . = 1M;
   .text BLOCK(4K) : ALIGN(4K)
   {
       *(.multiboot)
       *(.text)
   }
   .data : { *(.data) }
   .bss  : { *(.bss)  }
 }
```

Let's set the output format to be a 32-bit x86 executable. The entry point into this binary is the `start` function from your assembly file, which eventually loads the `main` program from the C file. Further down, this essentially is telling the linker how to merge your object code and at what offset. In the linker file, you explicitly specify the address in which to load your kernel binary. In this case, it is at 1M or a 1 megabyte offset. This is where the main kernel code is expected to be, and the bootloader will

find it here when it is time to load it.

Booting the Kernel

The most exciting part of the effort is that you can piggyback off the very popular GRand Unified Bootloader (GRUB) to load your kernel. In order to do this, you need to create a grub.cfg file. For the moment, write the following contents into a file of that name, and save it into your current working directory. When the time comes to build your ISO image, you'll install this file into its appropriate directory path.

grub.cfg

```
set timeout=3

menuentry "The Linux Journal Kernel" {
        multiboot /boot/kernel
}
```

Compilation Time

Build the `boot.asm` into an object file:

```
$ nasm -f elf32 boot.asm -o boot.o
```

Build the `kernel.c` into an object file:

```
$ gcc -m32 -c kernel.c -o kernel.o
```

Link both object files and create the final executable program (that is, your kernel):

```
$ ld -m elf_i386 -T linker.ld -o kernel boot.o kernel.o
```

Now, you should have a compiled file in the same working directory labeled `kernel`:

```
$ ls
```

```
boot.asm  boot.o  grub.cfg  kernel  kernel.c  kernel.o
 ↪linker.ld
```

This file is your kernel. You'll be booting into that kernel shortly.

Building a Bootable ISO Image

Create a staging environment with the following directory path (from your current working directory path):

```
$ mkdir -p iso/boot/grub
```

Let's double-check that the kernel is a multiboot file type (no output is expected with a return code of 0):

```
$ grub-file --is-x86-multiboot kernel
```

Now, copy the kernel into your iso/boot directory:

```
$ cp kernel iso/boot/
```

And, copy your grub.cfg into the iso/boot/grub directory:

```
$ cp grub.cfg iso/boot/grub/
```

Make the final ISO image pointing to your iso subdirectory in your current working directory path:

```
$ grub-mkrescue -o my-kernel.iso iso/
xorriso 1.4.8 : RockRidge filesystem manipulator,
 ↪libburnia project.

Drive current: -outdev 'stdio:my-kernel.iso'
Media current: stdio file, overwriteable
```

```
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data, 10.3g free
Added to ISO image: directory '/'='/tmp/grub.fqt0G4'
xorriso : UPDATE : 284 files added in 1 seconds
Added to ISO image: directory
 ↪'/'='/home/petros/devel/misc/kernel/iso'
xorriso : UPDATE : 288 files added in 1 seconds
xorriso : NOTE : Copying to System Area: 512 bytes from file
 ↪'/usr/lib/grub/i386-pc/boot_hybrid.img'
ISO image produced: 2453 sectors
Written to medium : 2453 sectors at LBA 0
Writing to 'stdio:my-kernel.iso' completed successfully.
```

Additional Notes

Say you want to expand on this tutorial by automating the entire process of building the final image. The best way to accomplish this is by throwing a Makefile into the project's root directory. Here's an example of what that Makefile would look like:

Makefile

```
CP := cp
RM := rm -rf
MKDIR := mkdir -pv

BIN = kernel
CFG = grub.cfg
ISO_PATH := iso
BOOT_PATH := $(ISO_PATH)/boot
GRUB_PATH := $(BOOT_PATH)/grub

.PHONY: all
all: bootloader kernel linker iso
    @echo Make has completed.
```

```
bootloader: boot.asm
    nasm -f elf32 boot.asm -o boot.o

kernel: kernel.c
    gcc -m32 -c kernel.c -o kernel.o

linker: linker.ld boot.o kernel.o
    ld -m elf_i386 -T linker.ld -o kernel boot.o kernel.o

iso: kernel
    $(MKDIR) $(GRUB_PATH)
    $(CP) $(BIN) $(BOOT_PATH)
    $(CP) $(CFG) $(GRUB_PATH)
    grub-file --is-x86-multiboot $(BOOT_PATH)/$(BIN)
    grub-mkrescue -o my-kernel.iso $(ISO_PATH)

.PHONY: clean
clean:
        $(RM) *.o $(BIN) *iso
```

To build (including the final ISO image), type:

```
$ make
```

To clean all of the build objects, type:

```
$ make clean
```

The Moment of Truth

You now have an ISO image, and if you did everything correctly, you should be able to boot into it from a CD on a physical machine or in a virtual machine (such as VirtualBox or QEMU). Start the virtual machine after configuring its profile to boot

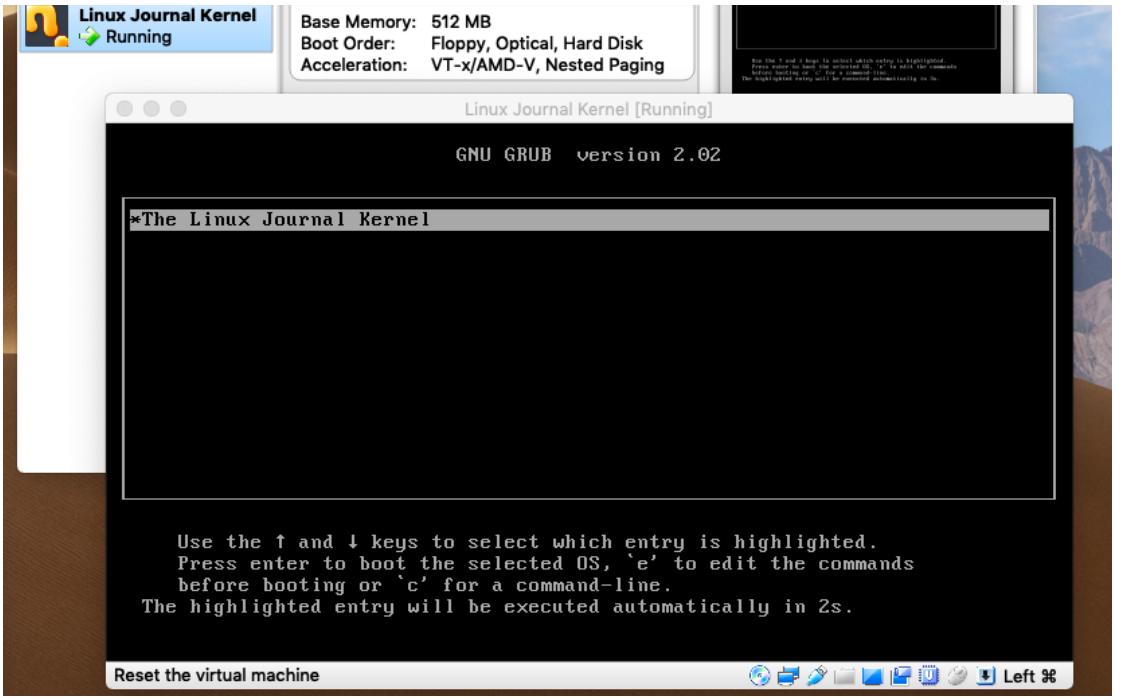


Figure 2. The GRUB Bootloader Counting Down to Load the Kernel

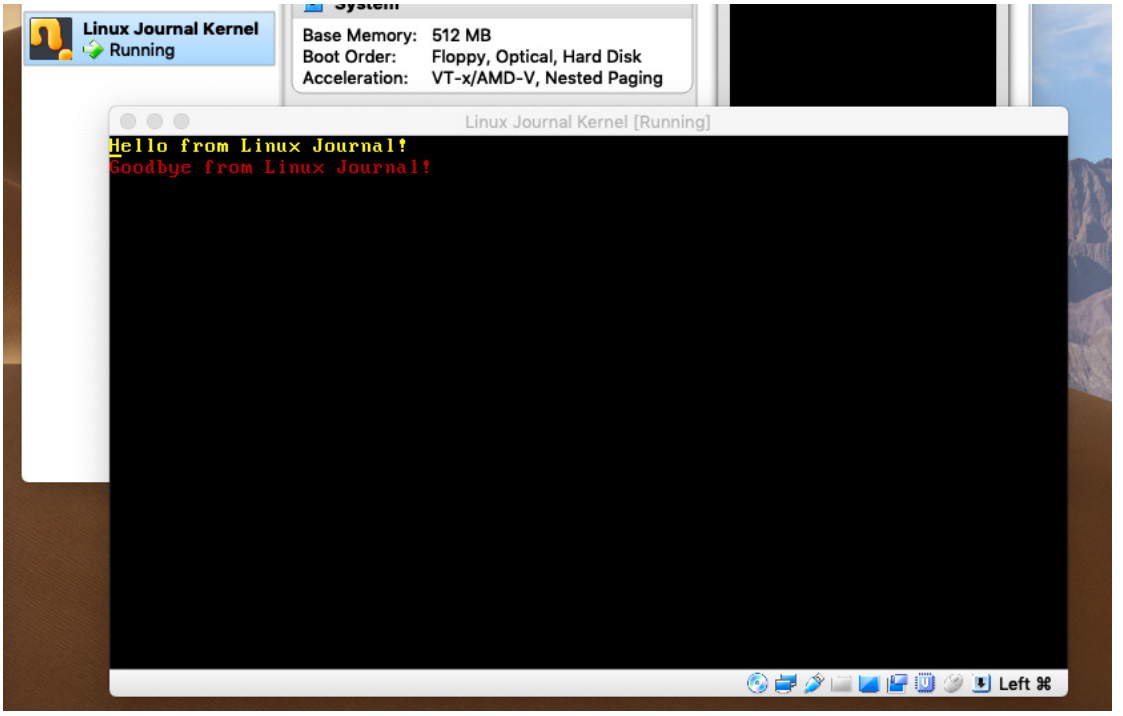


Figure 3. The *Linux Journal* kernel booted. Yes, it does only this.

from the ISO. You'll immediately be greeted by GRUB (Figure 2).

After the timeout elapses, the kernel will boot.

Summary

You did it! You wrote your very own kernel from scratch. Again, it doesn't do much of anything, but you definitely can expand upon this. Now, if you will excuse me, I need to post a message to the USENET newsgroup, *comp.os.minix*, about how I developed a new kernel, and that it *won't be big and professional like GNU*. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior performance software engineer at Cray for its Lustre High Performance File System division. He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Resources

- "DIY: Build a Custom Minimal Linux Distribution from Source" by Petros Koutoupis, *LJ*, June 2018
- "Build a Custom Minimal Linux Distribution from Source, Part II" by Petros Koutoupis, *LJ*, January 2019
- The Required Fields for the Multiboot Header
- The Computer Display Common Text Modes
- The Official NASM Project Page

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Memory Footprint of Processes

The amount of memory your system needs depends on the memory requirements of the programs you run. Do you want to know how to figure that out? It's not as simple as adding up the amount of memory used by each process individually, because some of that memory can be shared. Read on to learn the details.

By Frank Edwards

System administrators want to understand the applications that run on their systems. You can't tune a machine unless you know what the machine is doing! It's fairly easy to monitor a machine's physical resources: CPU (`mpstat`, `top`), memory (`vmstat`), disk IO (`iotop`, `blktrace`, `blkiomon`) and network bandwidth (`ip`, `nettop`).

Logical resources are just as important—if not more important—yet the tools to monitor them either don't exist or aren't exactly "user-friendly". For example, the `ps` command can report the RSS (resident set size) for a process. But how much of that is shared library and how much is application? Or executable code vs. data space? Those are questions that must be answered if a system administrator wants to calculate an application's memory footprint.

To answer these questions, and others, I describe extracting information from the /proc filesystem. First, let's look at terminology relevant to Linux memory management. If you want an exhaustive look at memory management on Linux, consider Mel Gorman's seminal work *Understanding the Linux Virtual Memory Manager*. His book is an oldie but goodie; the hardware he describes hasn't changed much over

the intervening years, and the changes that have occurred have been minor. This means the concepts he describes and much of the code used to implement those concepts is still spot-on.

Before going into the nuts and bolts of the answers to those questions, you first need to understand the context in which those questions are answered. So let's start with a high-level overview.

Linux Memory Usage

Your computer system has some amount of physical RAM installed. RAM is needed to run all software, because the CPU will fetch instructions and data from RAM and nowhere else. When a system doesn't have enough RAM to satisfy all processes, some of the process memory is written to an external storage device and that RAM then can be freed for use by other processes. This is called either *swapping*, when the RAM being freed is *anonymous memory* (meaning that it isn't associated with file data, such as shared memory or a process's heap space), or *paging* (which applies to things like memory-mapped files).

(By the way, a process is simply an application that's currently running. While the application is executing, it has a current directory, user and group credentials, a list of open files and network connections, and so on.)

Some types of memory don't need to be written out before they can be freed and reused. For example, the executable code of an application is stored in memory and protected as *read-only*. Since it can never be changed, when Linux wants to use that memory for something else, it just takes it! If the application ever needs that memory back again, Linux can reload it from the original application executable on disk. Also, since this memory is read-only, it can be used by multiple processes at the same time. And, this is where the confusion comes in regarding calculating how much memory a process is using—what if some of that memory is being shared with other processes? How do you account for it?

Before getting to that, I need to define a few other terms. The first is *pinned memory*.

Most memory is *pageable*, meaning that it can be swapped or paged out when the system is running low on RAM. But pinned memory is locked in place and can't be reused. This is obviously good for performance—the memory never can be taken away, so you never have to wait for it to be brought back in. The problem is that such memory can *never* be reused, even if the system is running critically low on RAM. Pinned memory reduces the system's flexibility when it comes to managing memory, and no one likes to be boxed into a corner.

Simple Example

I made reference above to read-only memory, memory that is shared, memory used for heap space, and so on. Below is some sample output that shows how memory is being used by my Bash shell (I want to emphasize that this output has been trimmed to fit into the allotted space, but all relevant fields are still represented. You can run the two commands you see on your own system and look at real data, if you wish. You'll see full pathnames instead of "..." as shown below, for example):

```
fedwards@local:~$ cd /proc/$$
fedwards@local:/proc/3867$ cat maps
00400000-004f4000 r-xp 00000000 08:01 260108 /bin/bash
006f3000-006f4000 r--p 000f3000 08:01 260108 /bin/bash
006f4000-006fd000 rw-p 000f4000 08:01 260108 /bin/bash
006fd000-00703000 rw-p 00000000 00:00 0
00f52000-01117000 rw-p 00000000 00:00 0      [heap]
f4715000-f4720000 r-xp 00000000 08:01 267196 /.../libnss_files-2.23.so
f4720000-f491f000 ---p 0000b000 08:01 267196 /.../libnss_files-2.23.so
f491f000-f4920000 r--p 0000a000 08:01 267196 /.../libnss_files-2.23.so
f4920000-f4921000 rw-p 0000b000 08:01 267196 /.../libnss_files-2.23.so
f4921000-f4927000 rw-p 00000000 00:00 0
f4f55000-f5914000 r--p 00000000 08:01 139223 /.../locale-archive
f6329000-f6330000 r--s 00000000 08:01 396945 /.../gconv-modules.cache
f6332000-f6333000 rw-p 00000000 00:00 0
fd827000-fd848000 rw-p 00000000 00:00 0      [stack]
fd891000-fd894000 r--p 00000000 00:00 0      [vvar]
```

```
fd894000-fd896000 r-xp 00000000 00:00 0      [vdso]
ff600000-ff601000 r-xp 00000000 00:00 0      [vsyscall]
fedwards@local:/proc/3867$
```

Each line of output represents one `vm_area`. A `vm_area` is a data structure inside the Linux kernel that keeps track of how one region of virtual memory is being used inside a process. The sample output has /bin/bash on the first three lines, because Linux has created three ranges of virtual memory that refer to the executable program. The first region has permissions `r-xp`, because it is executable code (r = read, x = execute and p = private; the dash means write permission is turned off). The second region refers to read-only data within the application and has permissions `r--p` (the two dashes represent write and execute permission).

The third region represents variables that have been given initial values in the application's source code, so it must be loaded from the executable, but it could be changed during runtime (hence the permissions `rw-p` that shows only execute is turned off). These regions can be any size, but they are made of up *pages*, which are each 4K on Linux. The term *page* means the smallest allocatable unit of virtual memory. (In technical documentation, you'll see two other terms: *frame* and *slot*. Frames and slots are the same size as pages, but frames refer to physical memory and slots refer to swap space.)

You know from my previous discussion that read-only regions are shared with other processes, so why does "p" show up in the permissions for the first region? Shouldn't it be a shared region? You have a good eye to spot that! Yes, it should. And in fact, it *is* shared. The reason it shows up as "p" here is because there are actually 14 different permissions and room only for four letters, so some condensing had to be done. The "p" means private, because while the memory is currently marked read-only, the application *could* change that permission and make it read-write, and if it did make that change and then modified the memory, you would not want other processes to see those changes! That would be similar to one process changing directory, and every other process on the system changing at the same time! Oops! So the letter "p" that marks the region as private really means *copy-on-write*. All of the memory

starts out being shared among all processes using that region, but if any part of it is modified in the future, that one tiny piece is copied into another part of RAM so that the change applies only to the one process that attempted the write. In essence, it's private, even though 99% of the time, the memory in that region will be shared with other processes. Such copying applies on a page-by-page basis, not the entire `vm_area`. Now you can begin to see the difficulty in calculating how much memory a process actually consumes.

But while I'm on this topic, there's a region in the list that has an "s" in the permission field. That region is a *memory-mapped file*, meaning that the data blocks on disk are mapped to the virtual memory addresses shown in the listing. Any reference the process makes to the memory addresses are translated automatically into reads and writes to the corresponding data blocks on disk. The memory used by this region is actually shared by all processes that map the file into memory, meaning no duplicated memory for file access by those processes.

Just because a region represents some given size of virtual memory does not necessarily mean that there are physical frames of RAM for every virtual page. In fact, this is often the case. Imagine an application that allocates 100MB of memory. Should the operating system actually allocate 100MB right then? UNIX systems do not—they allocate a region of virtual memory like those above, but no physical RAM. As the process tries to access those virtual addresses, page faults will be generated, and the operating system will allocate the memory at that time. Deferring memory allocation until the last possible moment is one way that Linux optimizes the use of memory, but it complicates the task in trying to determine how much memory an application is using.

Recap So Far

A process's address space is broken up into regions called `vm_areas`. These `vm_areas` are unique to each process, but the frames of memory referred to by the pages within the `vm_area` might be shared across processes. If the memory is read-only (like executable code), all processes share the frame equally. Any attempt to write to virtual pages that are read-only triggers a page fault that is converted into a SIGSEGV and the process is killed. (You may have seen the message pop up on your terminal

screen, “Segmentation fault.” That means the process was killed by SIGSEGV.)

Memory that is read/write also can be shared, such as shared memory. If multiple processes can write to the frames of the `vm_area` equally, some form of synchronization inside the application will be necessary, or multiple processes could write at the same time, possibly corrupting the contents of that shared memory. (Most applications will use some kind of mutex lock for this, but synchronization and locking is outside the scope of this article.)

Adding Up the Memory Actually Used

So, determining how much memory a process consumes is difficult. You could add up the space allocated to the `vm_areas`, but that’s virtual memory, not physical; large portions of that space could be unused or swapped out. This number is not a true representation of the amount of memory being used by the process.

You could add up only the frames that are used by this process and not shared. (This information is available in /proc/pid/smaps.) You might call this the “USS” (Unique Set Size), as it defines how much memory will be freed when an application terminates (shared libraries typically stay in RAM even when no processes are currently using them as a performance optimization for when they are needed again). But this isn’t the true memory cost of a process either, as the process likely uses one or more shared libraries. For example, if an application is executed and it uses a shared library that isn’t already in memory, that library must be loaded—some part of that library should be allocated against the new process, right?

The `ps` command reports the “RSS” (Resident Set Size), which includes *all* frames used by the process, regardless of whether they’re shared. Unfortunately, this number is going to inflate the memory size when all processes are summed up—adding up this number for all processes running on the system will count all shared libraries multiple times, greatly inflating the actual memory requirement.

The /proc/pid/smaps file includes yet another memory category, PSS (Proportional Set Size). This is the amount of unique memory just for one process (the USS), plus a

proportion of the memory that is shared by other running processes. For example, let's assume the USS for a process is 2MB and it uses another 4MB of shared libraries, but those shared libraries are used by three other processes. Since there are four processes using the shared libraries, they should each only be accounted for 25% of the overall library size. That would make the PSS of the process 2MB + (4MB / 4) = 3MB. If you now add together the PSS values of all processes on the system, the shared library memory will be totally accounted for, meaning the whole is equal to the sum of its parts.

It's not perfect—when one of those processes terminates, the memory returned to the system will be USS, and because there's one less process using the shared libraries, the PSS of all other processes will appear to increase! A naïve system administrator might wonder why the memory usage on the remaining processes has suddenly spiked, but in truth, it hasn't. In this example, 4MB/4 becomes 4MB/3, so any process using the shared libraries will see an adjusted PSS value that goes up by .33MB.

As the last step, I'm going to demonstrate a command that performs these calculations.

Automating the Work

The one-line command shown below will accumulate all of the PSS values for all processes on the system:

```
awk '/^Pss:/ { ttl += $2 }; END { print ttl }' /proc/[0-9]*/smaps
 ↪2>/dev/null
```

Note that stderr is redirected to /dev/null. This is because the shell replaces the wildcard string with a list of all filenames that match and then executes the `awk` command. This means that by the time the `awk` command is running, some of those processes already may have terminated. That will cause `awk` to print an error message about a non-existent file, hence redirecting stderr to avoid that error. (Astute readers will note that this command line will *never* factor in the memory consumed by the `awk` command itself!)

Many of the processes that the `awk` command is going to be reading will not be accessible to an unprivileged account, so system administrators should consider using `sudo` to run the

command. (Inaccessible processes will produce error messages that are then redirected to /dev/null, thus the command will report a total of the memory used by all processes that *are* accessible—in other words, those owned by the current user.)

Summary

I've covered a lot of ground in this article, from terminology (pages, frames, slots) and background information on how virtual memory is organized (`vm_areas`), to details on how memory usage is reported to userspace (the maps and smaps files under /proc). I've barely scratched the surface of the type of information that the Linux kernel exposes to userspace, but hopefully, this has piqued your interest enough that you'll explore it further. ■

Frank Edwards has been a programmer since the days of the Zilog Z-80 in the TRS-80 computer, circa 1978. For some people, programming is a hobby, or a job or a career—for him, it's an obsession. He once disassembled an operating system just to see how it worked. Most of his early life was spent in C, but he has branched out considerably since then (Java, Python, Perl, Swift and UNIX shell being where he spends most of his time).

Resources

My favorite source for technical details is LWN.net if I'm looking for discussion and analysis, but I frequently will go straight to the Linux source code when I'm looking for implementation details. See "ELC: How much memory are applications really using?" for the discussion around adding PSS to smaps, and see "Tracking actual memory utilization" for a discussion of memory used by a process but that belongs to the kernel (something this article doesn't touch upon).

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Oops! Debugging Kernel Panics

A look into what causes kernel panics and some utilities to help gain more information.

By Petros Koutoupis

Working in a Linux environment, how often have you seen a kernel panic? When it happens, your system is left in a crippled state until you reboot it completely. And, even after you get your system back into a functional state, you're still left with the

```
[   27.509349] CR2: 0000000000000000 CR3: 0000000119506000 CR4: 0000000000060670
[   27.509938] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[   27.510500] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[   27.511080] Stack:
[   27.511518]  ffffffffa702c597 0000000000000002 fffffffffffffffb ffffbca4008b3
f08
[   27.513069]  0000000002375008 ffffffffa702c9db ffff9725965ac000 ffffffffa6e7a
c00
[   27.514603]  0000000000000002 ffff972596333f00 ffffffffa6e0b3b0 ffff972596333
f00
[   27.516136] Call Trace:
[   27.516569]  [<ffffffffa702c597>] ? __handle_sysrq+0xf7/0x150
[   27.517063]  [<ffffffffa702c9db>] ? write_sysrq_trigger+0x2b/0x30
[   27.517565]  [<ffffffffa6e7ac00>] ? proc_reg_write+0x40/0x70
[   27.518057]  [<ffffffffa6e0b3b0>] ? vfs_write+0xb0/0x190
[   27.518540]  [<ffffffffa6e0c7f2>] ? SyS_write+0x52/0xc0
[   27.519037]  [<ffffffffa6c03b7d>] ? do_syscall_64+0x8d/0xf0
[   27.519530]  [<ffffffffa721924e>] ? entry_SYSCALL_64_after_swapgs+0x58/0xc6
[   27.520063] Code: 41 5c 41 5d 41 5e 41 5f e9 3c 08 cf ff 66 2e 0f 1f 84 00 00
 00 00 00 66 90 0f 1f 44 00 00 c7 05 29 5e a8 00 01 00 00 00 0f ae f8 <c6> 04 25
 00 00 00 00 01 c3 0f 1f 44 00 00 0f 1f 44 00 00 53 8d
[   27.533456] RIP  [<ffffffffa702be62>] sysrq_handle_crash+0x12/0x20
[   27.534112]  RSP <ffffbca4008b3e78>
[   27.534538] CR2: 0000000000000000
```

Figure 1. A Typical Kernel Panic

question: why? You may have no idea what happened or why it happened. Those questions can be answered though, and the following guide will help you root out the cause of some of the conditions that led to the original crash.

Let's start by looking at a set of utilities known as `kexec` and `kdump`. `kexec` allows you to boot into another kernel from an existing (and running) kernel, and `kdump` is a `kexec`-based crash-dumping mechanism for Linux.

Installing the Required Packages

First and foremost, your kernel should have the following components statically built in to its image:

```
CONFIG_RELOCATABLE=y
CONFIG_KEXEC=y
CONFIG_CRASH_DUMP=y
CONFIG_DEBUG_INFO=y
CONFIG_MAGIC_SYSRQ=y
CONFIG_PROC_VMCORE=y
```

You can find this in `/boot/config-'uname -r'`.

Make sure that your operating system is up to date with the latest-and-greatest package versions:

```
$ sudo apt update && sudo apt upgrade
```

Install the following packages (I'm currently using Debian, but the same should and will apply to Ubuntu):

```
$ sudo apt install gcc make binutils linux-headers-'uname -r'
 ↪kdump-tools crash 'uname -r'-dbg
```

Note: Package names may vary across distributions.

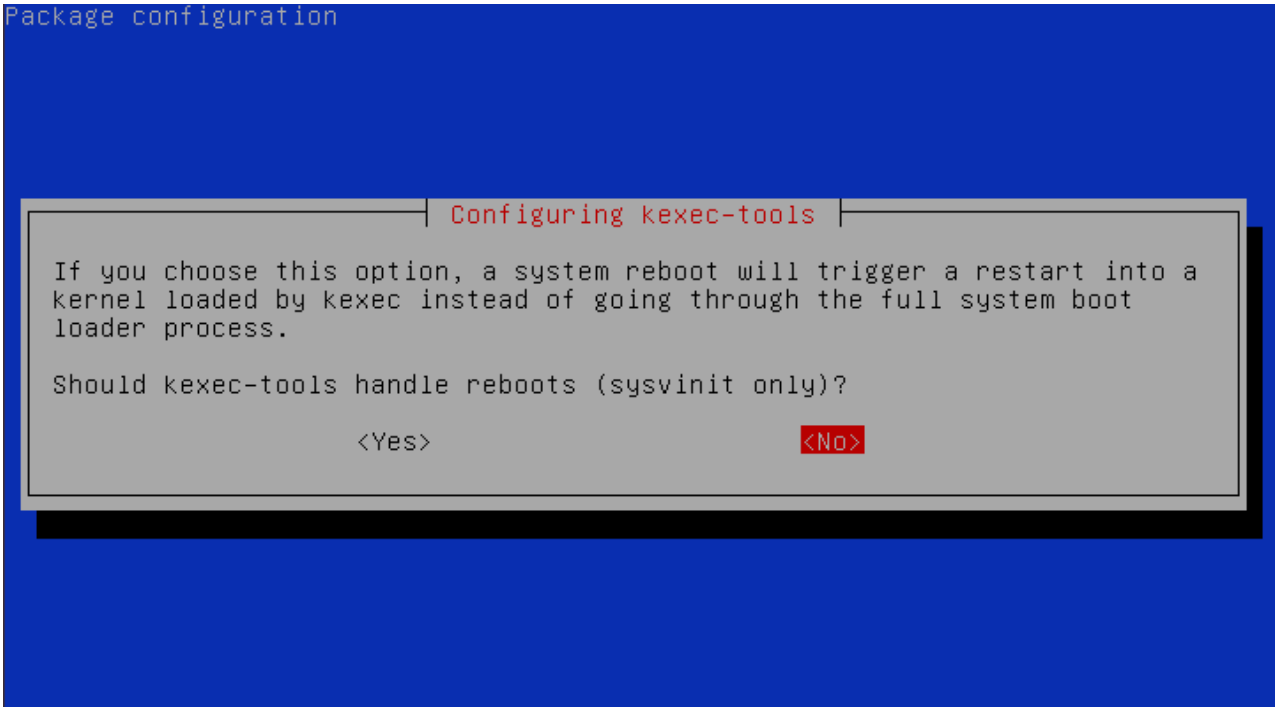


Figure 2. kexec Configuration Menu

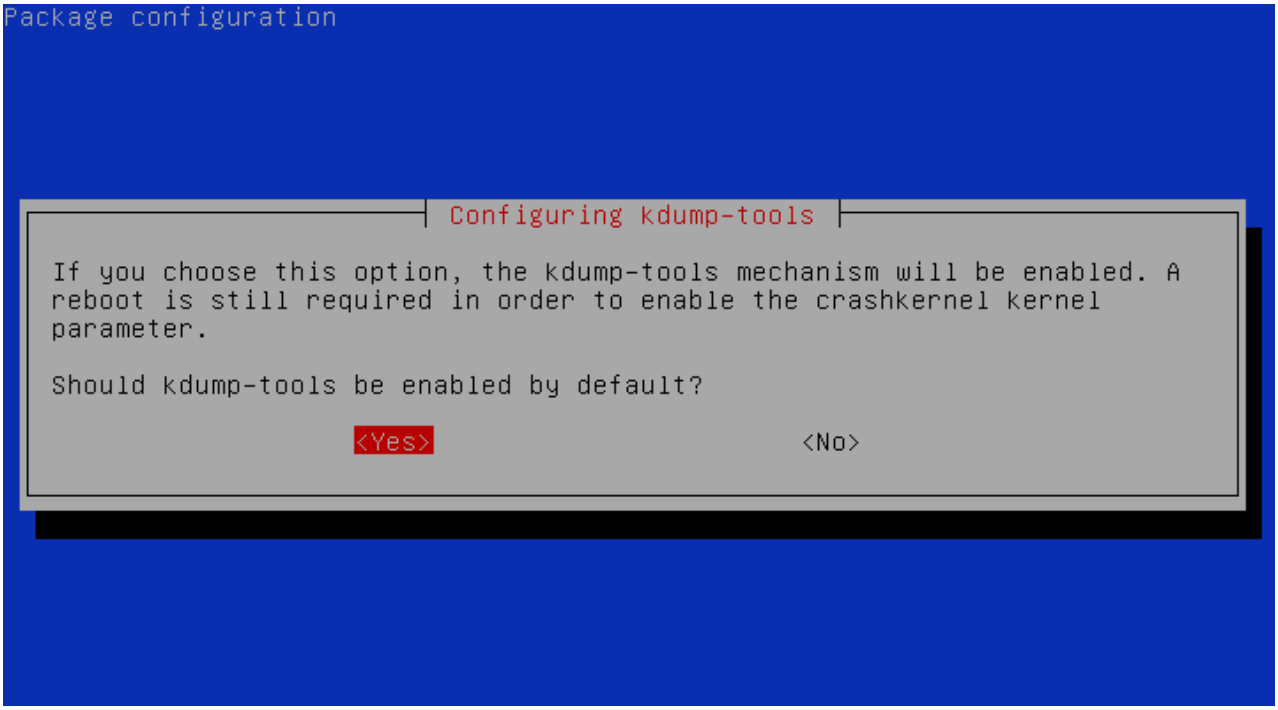


Figure 3. kdump Configuration Menu

During the installation, you will be prompted with questions to enable `kexec` to handle reboots (answer whatever you'd like, but I answered "no"; see Figure 2).

And to enable `kdump` to run and load at system boot, answer "yes" (Figure 3).

Configuring kdump

Open the /etc/default/kdump-tools file, and at the very top, you should see the following:

```
USE_KDUMP=1
#KDUMP_SYSCTL="kernel.panic_on_oops=1"
```

Eventually, you'll write a custom module that will trigger an OOPS kernel condition, and in order to have `kdump` gather and save the state of the system for post-mortem analysis, you'll need to enable your kernel to panic on this OOPS condition. In order to do this, uncomment the line that starts with `KDUMP_SYSCTL`:

```
USE_KDUMP=1
KDUMP_SYSCTL="kernel.panic_on_oops=1"
```

The initial testing will require that SysRq be enabled. There are a few ways to do that, but here I provide instructions to enable support for this feature on system reboot. Open the /etc/sysctl.d/99-sysctl.conf file, and make sure that the following line (closer to the bottom of the file) is uncommented:

```
kernel.sysrq=1
```

Now, open this file: /etc/default/grub.d/kdump-tools.default. You will find a single line that looks like this:

```
GRUB_CMDLINE_LINUX_DEFAULT="$GRUB_CMDLINE_LINUX_DEFAULT
 ↪crashkernel=384M-:128M"
```

Modify the section that reads `crashkernel=384M-:128M` to `crashkernel=128M`.

Now, update your GRUB boot configuration file:

```
$ sudo update-grub
[sudo] password for petros:
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.9.0-8-amd64
Found initrd image: /boot/initrd.img-4.9.0-8-amd64
done
```

And, reboot the system.

Verifying Your kdump Environment

After coming back from the reboot, `dmesg` will log the following:

```
$ sudo dmesg |grep -i crash
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-8-amd64
 ↪root=UUID=bd76b0fe-9d09-40a9-a0d8-a7533620f6fa ro quiet
 ↪crashkernel=128M
[    0.000000] Reserving 128MB of memory at 720MB for crashkernel
 ↪(System RAM: 4095MB)
[    0.000000] Kernel command line: BOOT_IMAGE=/boot/
↪vmlinuz-4.9.0-8-amd64
 ↪root=UUID=bd76b0fe-9d09-40a9-a0d8-a7533620f6fa ro
 ↪quiet crashkernel=128M
```

While your kernel will have the following features enabled (a "1" means enabled):

```
$ sudo sysctl -a|grep kernel|grep -e panic_on_oops -e sysrq
kernel.panic_on_oops = 1
kernel.sysrq = 1
```

Your `kdump` service should be running:

```
$ sudo systemctl status kdump-tools.service
 kdump-tools.service - Kernel crash dump capture service
   Loaded: loaded (/lib/systemd/system/kdump-tools.service;
    ↪enabled; vendor preset: enabled)
   Active: active (exited) since Tue 2019-02-26 08:13:34 CST;
    ↪1h 33min ago
  Process: 371 ExecStart=/etc/init.d/kdump-tools start
   ↪(code=exited, status=0/SUCCESS)
 Main PID: 371 (code=exited, status=0/SUCCESS)
    Tasks: 0 (limit: 4915)
   CGroup: /system.slice/kdump-tools.service

Feb 26 08:13:34 deb-panic systemd[1]: Starting Kernel crash
 ↪dump capture service...
Feb 26 08:13:34 deb-panic kdump-tools[371]: Starting
 ↪kdump-tools: loaded kdump kernel.
Feb 26 08:13:34 deb-panic kdump-tools[505]: /sbin/kexec -p
 ↪--command-line="BOOT_IMAGE=/boot/vmlinuz-4.9.0-8-amd64 root=
Feb 26 08:13:34 deb-panic kdump-tools[506]: loaded kdump kernel
Feb 26 08:13:34 deb-panic systemd[1]: Started Kernel crash dump
 ↪capture service.
```

Your crash kernel should be loaded (into memory and in the 128M region you defined earlier):

```
$ cat /sys/kernel/kexec_crash_loaded
1
```

You can verify your `kdump` configuration further here:

```
$ sudo kdump-config show
DUMP_MODE:        kdump
USE_KDUMP:        1
```

```
KDUMP_SYSCTL:     kernel.panic_on_oops=1
KDUMP_COREDIR:    /var/crash
crashkernel addr: 0x2d000000
   /var/lib/kdump/vmlinuz: symbolic link to /boot/
↪vmlinuz-4.9.0-8-amd64
kdump initrd:
   /var/lib/kdump/initrd.img: symbolic link to /var/lib/kdump/
↪initrd.img-4.9.0-8-amd64
current state:    ready to kdump

kexec command:
  /sbin/kexec -p --command-line="BOOT_IMAGE=/boot/
↪vmlinuz-4.9.0-8-amd64 root=UUID=bd76b0fe-9d09-40a9-
↪a0d8-a7533620f6fa ro quiet irqpoll nr_cpus=1 nousb
 ↪systemd.unit=kdump-tools.service
 ↪ata_piix.prefer_ms_hyperv=0"
 ↪--initrd=/var/lib/kdump/initrd.img /var/lib/kdump/vmlinuz
```

Let's also test it without actually running it:

```
$ sudo kdump-config test
USE_KDUMP:        1
KDUMP_SYSCTL:     kernel.panic_on_oops=1
KDUMP_COREDIR:    /var/crash
crashkernel addr:  0x2d000000
kdump kernel addr:
kdump kernel:
   /var/lib/kdump/vmlinuz: symbolic link to /boot/
↪vmlinuz-4.9.0-8-amd64
kdump initrd:
   /var/lib/kdump/initrd.img: symbolic link to
 ↪/var/lib/kdump/initrd.img-4.9.0-8-amd64
kexec command to be used:
```

```
  /sbin/kexec -p --command-line="BOOT_IMAGE=/boot/
↪vmlinuz-4.9.0-8-amd64 root=UUID=bd76b0fe-9d09-40a9-
↪a0d8-a7533620f6fa ro quiet irqpoll nr_cpus=1 nousb
 ↪systemd.unit=kdump-tools.service
 ↪ata_piix.prefer_ms_hyperv=0"
 ↪--initrd=/var/lib/kdump/initrd.img /var/lib/kdump/vmlinuz
```

The Moment of Truth

Now that your environment is loaded to make use of `kdump`, you probably should test it, and the best way to test it is by forcing a kernel crash over SysRq. Assuming your kernel is built with SysRq support, crashing a running kernel is as simple as typing:

```
$ echo "c" | sudo tee -a /proc/sysrq-trigger
```

What should you expect? You'll see a kernel panic/crash similar to the one shown in Figure 1. Following this crash, the kernel loaded over kexec will collect the state of the system, which includes everything relevant in memory, on the CPU, in dmesg, in loaded modules and more. It then will save this valuable crash data somewhere in /var/crash for further analysis. Once the collection of information completes, the system will reboot automatically and will bring you back to a functional state.

What Now?

You now have your crash file, and again, it's located in /var/crash:

```
$ cd /var/crash/
$ ls
201902261006  kexec_cmd
$ cd 201902261006/
```

Although before opening the crash file, you probably should install the kernel's source package:

```
$ sudo apt source linux-image-'uname -r'
```

Earlier, you installed a debug version of your Linux kernel containing the unstripped debug symbols required for this type of debugging analysis. Now you need that kernel. Open the kernel crash file with the `crash` utility:

```
$ sudo crash dump.201902261006 /usr/lib/debug/
↪vmlinux-4.9.0-8-amd64
```

Once everything loads, a summary of the panic will appear on the screen:

```
      KERNEL: /usr/lib/debug/vmlinux-4.9.0-8-amd64
    DUMPFILE: dump.201902261006  [PARTIAL DUMP]
        CPUS: 4
        DATE: Tue Feb 26 10:07:21 2019
      UPTIME: 00:04:09
LOAD AVERAGE: 0.00, 0.00, 0.00
       TASKS: 100
    NODENAME: deb-panic
     RELEASE: 4.9.0-8-amd64
     VERSION: #1 SMP Debian 4.9.144-3 (2019-02-02)
     MACHINE: x86_64  (2592 Mhz)
      MEMORY: 4 GB
       PANIC: "sysrq: SysRq : Trigger a crash"
         PID: 563
     COMMAND: "tee"
        TASK: ffff88e69628c080 [THREAD_INFO: ffff88e69628c080]
         CPU: 2
       STATE: TASK_RUNNING (SYSRQ)
```

Notice the reason for the panic: `sysrq: SysRq : Trigger a crash`. Also, notice the command that led to it: `tee`. None of this should be a surprise since you triggered it.

If you run a backtrace of what the kernel functions were that led to the panic, you

should see the following (processed by CPU core no. 2):

```
crash> bt
PID: 563    TASK: ffff88e69628c080  CPU: 2   COMMAND: "tee"
 #0 [ffffa67440b23ba0] machine_kexec at ffffffffa0c53f68
 #1 [ffffa67440b23bf8] __crash_kexec at ffffffffa0d086d1
 #2 [ffffa67440b23cb8] crash_kexec at ffffffffa0d08738
 #3 [ffffa67440b23cd0] oops_end at ffffffffa0c298b3
 #4 [ffffa67440b23cf0] no_context at ffffffffa0c619b1
 #5 [ffffa67440b23d50] __do_page_fault at ffffffffa0c62476
 #6 [ffffa67440b23dc0] page_fault at ffffffffa121a618
    [exception RIP: sysrq_handle_crash+18]
    RIP: ffffffffa102be62  RSP: ffffa67440b23e78  RFLAGS: 00010282
    RAX: ffffffffa102be50  RBX: 0000000000000063  RCX: 0000000000000000
    RDX: 0000000000000000  RSI: ffff88e69fd10648  RDI: 0000000000000063
    RBP: ffffffffa18bf320   R8: 0000000000000001   R9: 0000000000007eb8
    R10: 0000000000000001  R11: 0000000000000001  R12: 0000000000000004
    R13: 0000000000000000  R14: 0000000000000000  R15: 0000000000000000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #7 [ffffa67440b23e78] __handle_sysrq at ffffffffa102c597
 #8 [ffffa67440b23ea0] write_sysrq_trigger at ffffffffa102c9db
 #9 [ffffa67440b23eb0] proc_reg_write at ffffffffa0e7ac00
#10 [ffffa67440b23ec8] vfs_write at ffffffffa0e0b3b0
#11 [ffffa67440b23ef8] sys_write at ffffffffa0e0c7f2
#12 [ffffa67440b23f38] do_syscall_64 at ffffffffa0c03b7d
#13 [ffffa67440b23f50] entry_SYSCALL_64_after_swapgs at ffffffffa121924e
    RIP: 00007f3952463970  RSP: 00007ffc7f3a4e58  RFLAGS: 00000246
    RAX: ffffffffffffffda  RBX: 0000000000000002  RCX: 00007f3952463970
    RDX: 0000000000000002  RSI: 00007ffc7f3a4f60  RDI: 0000000000000003
    RBP: 00007ffc7f3a4f60   R8: 00005648f508b610   R9: 00007f3952944480
    R10: 0000000000000839  R11: 0000000000000246  R12: 0000000000000002
    R13: 0000000000000001  R14: 00005648f508b530  R15: 0000000000000002
    ORIG_RAX: 0000000000000001  CS: 0033  SS: 002b
```

In your backtrace, you should notice the symbol address of what is stored in your Return Instruction Pointer (RIP): `ffffffffa102be62`. Let's take a look at this symbol address:

```
crash> sym ffffffffa102be62
ffffffffa102be62 (t) sysrq_handle_crash+18 ./debian/build/
↪build_amd64_none_amd64/./drivers/tty/sysrq.c: 144
```

Wait a minute! The exception seems to have been triggered in line 144 of the drivers/tty/sysrq.c file *and* inside the `sysrq_handle_crash` function. Hmm…I wonder what's happening in this kernel source file. (This is why I had you installed your kernel source package moments ago.) Let's navigate to the /usr/src directory and untar the source package:

```
$ cd /usr/src
$ ls
linux_4.9.144-3.debian.tar.xz  linux_4.9.144.orig.tar.xz
 ↪linux-headers-4.9.0-8-common
linux_4.9.144-3.dsc            linux-headers-4.9.0-8-amd64
 ↪linux-kbuild-4.9
$ sudo tar xJf linux_4.9.144.orig.tar.xz
$ vim linux-4.9.144/drivers/tty/sysrq.c
```

Locate the `sysrq_handle_crash` function:

```
static void sysrq_handle_crash(int key)
{
    char *killer = NULL;

    /* we need to release the RCU read lock here,
     * otherwise we get an annoying
     * 'BUG: sleeping function called from invalid context'
     * complaint from the kernel before the panic.
```

```
     */
    rcu_read_unlock();
    panic_on_oops = 1;      /* force panic */
    wmb();
    *killer = 1;
}
```

And more specifically, look at line 144:

```
*killer = 1;
```

It was this line that led to the page fault logged in line #6 of the backtrace:

```
#6 [ffffa67440b23dc0] page_fault at ffffffffa121a618
```

Okay. So, now you should have a basic understanding of how to debug bad kernel code, but what happens if you want to debug your very own custom kernel modules (for example, drivers)? I wrote a simple Linux kernel module that essentially invokes a similar style of a kernel crash when loaded. Call it test-module.c and save it somewhere in your home directory:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/version.h>

static int test_module_init(void)
{
        int *p = 1;
printk("%d\n", *p);
        return 0;
}
static void test_module_exit(void)
{
```

```
        return;
}

module_init(test_module_init);
module_exit(test_module_exit);
```

You'll need a Makefile to compile this kernel module (save it in the same directory):

```
obj-m += test-module.o

all:
    $(MAKE) -C/lib/modules/$(shell uname -r)/build M=$(PWD)
```

Run the `make` command to compile the module and do *not* delete any of the compilation artifacts; you'll need those later:

```
$ make
make -C/lib/modules/4.9.0-8-amd64/build M=/home/petros
make[1]: Entering directory '/usr/src/
↪linux-headers-4.9.0-8-amd64'
  CC [M]  /home/petros/test-module.o
/home/petros/test-module.c: In function "test_module_init":
/home/petros/test-module.c:7:11: warning: initialization makes
 ↪pointer from integer without a cast [-Wint-conversion]
  int *p = 1;
          ^
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /home/petros/test-module.ko
make[1]: Leaving directory '/usr/src/
↪linux-headers-4.9.0-8-amd64'
```

Note: you may see a compilation warning. Ignore it for now. This warning will be what

triggers your kernel crash.

Be careful now. Once you load the .ko file, the system will crash, so make sure everything is saved and synchronized to disk:

```
$ sync && sudo insmod test-module.ko
```

Similar to before, the system will crash, the `kexec` kernel/environment will help gather everything and save it somewhere in /var/crash, followed by an automatic reboot. After you have rebooted and are back into a functional state, locate the new crash directory and change into it:

```
$ cd /var/crash/201902261035/
```

Also, copy the unstripped kernel object file for your `test-module` from your home directory and into the current working directory:

```
$ sudo cp ~/test.o /var/crash/201902261035/
```

Load the crash file with your debug kernel:

```
$ sudo crash dump.201902261035 /usr/lib/debug/
↪vmlinux-4.9.0-8-amd64
```

Your summary should look something like this:

```
      KERNEL: /usr/lib/debug/vmlinux-4.9.0-8-amd64
    DUMPFILE: dump.201902261035  [PARTIAL DUMP]
        CPUS: 4
        DATE: Tue Feb 26 10:37:47 2019
      UPTIME: 00:11:16
LOAD AVERAGE: 0.24, 0.06, 0.02
       TASKS: 102
```

```
    NODENAME: deb-panic
     RELEASE: 4.9.0-8-amd64
     VERSION: #1 SMP Debian 4.9.144-3 (2019-02-02)
     MACHINE: x86_64  (2592 Mhz)
      MEMORY: 4 GB
       PANIC: "BUG: unable to handle kernel NULL pointer
↪dereference at 0000000000000001"
         PID: 1493
     COMMAND: "insmod"
        TASK: ffff893c5a5a5080 [THREAD_INFO: ffff893c5a5a5080]
         CPU: 3
       STATE: TASK_RUNNING (PANIC)
```

The reason for the kernel crash is summarized as follows: `BUG: unable to handle kernel NULL pointer dereference at 0000000000000001`. The userspace command that led to the panic was your `insmod`.

A backtrace will reveal a page fault exception at address `ffffffffc05ed005`:

```
crash> bt
PID: 1493   TASK: ffff893c5a5a5080  CPU: 3  COMMAND: "insmod"
 #0 [ffff9dcd013b79f0] machine_kexec at ffffffffa3a53f68
 #1 [ffff9dcd013b7a48] __crash_kexec at ffffffffa3b086d1
 #2 [ffff9dcd013b7b08] crash_kexec at ffffffffa3b08738
 #3 [ffff9dcd013b7b20] oops_end at ffffffffa3a298b3
 #4 [ffff9dcd013b7b40] no_context at ffffffffa3a619b1
 #5 [ffff9dcd013b7ba0] __do_page_fault at ffffffffa3a62476
 #6 [ffff9dcd013b7c10] page_fault at ffffffffa401a618
    [exception RIP: init_module+5]
    RIP: ffffffffc05ed005  RSP: ffff9dcd013b7cc8  RFLAGS: 00010246
    RAX: 0000000000000000  RBX: 0000000000000000  RCX: 0000000000000000
    RDX: 0000000080000000  RSI: ffff893c5a5a5ac0  RDI: ffffffffc05ed000
    RBP: ffffffffc05ed000   R8: 0000000000020098   R9: 0000000000000006
```

```
    R10: 0000000000000000  R11: ffff893c5a4d8100  R12: ffff893c5880d460
    R13: ffff893c56500e80  R14: ffffffffc05ef000  R15: ffffffffc05ef050
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #7 [ffff9dcd013b7cc8] do_one_initcall at ffffffffa3a0218e
 #8 [ffff9dcd013b7d38] do_init_module at ffffffffa3b81531
 #9 [ffff9dcd013b7d58] load_module at ffffffffa3b04aaa
#10 [ffff9dcd013b7e90] SYSC_finit_module at ffffffffa3b051f6
#11 [ffff9dcd013b7f38] do_syscall_64 at ffffffffa3a03b7d
#12 [ffff9dcd013b7f50] entry_SYSCALL_64_after_swapgs at ffffffffa401924e
    RIP: 00007f124662c469  RSP: 00007fffc4ca04a8  RFLAGS: 00000246
    RAX: ffffffffffffffda  RBX: 0000564213d111f0  RCX: 00007f124662c469
    RDX: 0000000000000000  RSI: 00005642129d3638  RDI: 0000000000000003
    RBP: 00005642129d3638   R8: 0000000000000000   R9: 00007f12468e3ea0
    R10: 0000000000000003  R11: 0000000000000246  R12: 0000000000000000
    R13: 0000564213d10130  R14: 0000000000000000  R15: 0000000000000000
    ORIG_RAX: 0000000000000139  CS: 0033  SS: 002b
```

Let's attempt to look at the symbol at the address `ffffffffc05ed005`:

```
crash> sym ffffffffc05ed005
ffffffffc05ed005 (t) init_module+5 [test_module]
```

Hmm. The issue occurred somewhere in the module initialization code of the `test_module` kernel driver. But what happened to all of the details shown in the earlier analysis? Well, because this code is not part of the debug kernel image, you'll need to find a way to load it into your crash analysis. This is why I instructed you to copy over the unstripped object file into your current working directory. Now it's time to load the module's object file:

```
crash> mod -s test ./test.o
     MODULE       NAME                    SIZE  OBJECT FILE
ffffffffc05ef000  test                   16384  ./test.o
```

Now you can go back and look at the same symbol address:

```
crash> sym ffffffffc05ed005
ffffffffc05ed005 (T) init_module+5 [test-module]
 ↪/home/petros/test-module.c: 8
```

And, now it's time to revisit to your code and look at line 8:

```
$ sed -n 8p test.c
        printk("%d\n", *p);
```

There you have it. The page fault occurred when you attempted to print the poorly defined pointer. Remember the compilation warning from earlier? Well, it was warning you for a reason, and in this current case, it's the reason that triggered the kernel panic. You may not be as fortunate in future coding cases.

What Else Can You Do Here?

The kernel crash file will preserve many artifacts from your system at the event of your crash. You can list a short summary of available commands with the `help` command:

```
crash> help

*           files       mach        repeat      timer
alias       foreach     mod         runq        tree
ascii       fuser       mount       search      union
bt          gdb         net         set         vm
btop        help        p           sig         vtop
dev         ipcs        ps          struct      waitq
dis         irq         pte         swap        whatis
eval        kmem        ptob        sym         wr
exit        list        ptov        sys         q
extend      log         rd          task
```

For instance, if you want to see a general summary of memory utilization:

```
crash> kmem -i
                 PAGES        TOTAL      PERCENTAGE
    TOTAL MEM   979869       3.7 GB         ----
         FREE   835519       3.2 GB   85% of TOTAL MEM
         USED   144350     563.9 MB   14% of TOTAL MEM
       SHARED     8374      32.7 MB    0% of TOTAL MEM
      BUFFERS     3849        15 MB    0% of TOTAL MEM
       CACHED        0            0    0% of TOTAL MEM
         SLAB     5911      23.1 MB    0% of TOTAL MEM

   TOTAL SWAP  1047807         4 GB         ----
    SWAP USED        0            0    0% of TOTAL SWAP
    SWAP FREE  1047807         4 GB  100% of TOTAL SWAP

 COMMIT LIMIT  1537741       5.9 GB         ----
    COMMITTED    16370      63.9 MB    1% of TOTAL LIMIT
```

If you want to see what `dmesg` logged up to the point of the failure:

```
crash> log

[    0.000000] Linux version 4.9.0-8-amd64
 ↪(debian-kernel@lists.debian.org) (gcc version 6.3.0
 ↪20170516 (Debian 6.3.0-18+deb9u1) ) #1 SMP Debian
 ↪4.9.144-3 (2019-02-02)
[    0.000000] Command line: BOOT_IMAGE=/boot/
↪vmlinuz-4.9.0-8-amd64 root=UUID=bd76b0fe-9d09-40a9-
↪a0d8-a7533620f6fa ro quiet crashkernel=128M
[    0.000000] x86/fpu: Supporting XSAVE feature 0x001:
 ↪'x87 floating point registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x002:
```

```
 ↪'SSE registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x004:
 ↪'AVX registers'
[    0.000000] x86/fpu: xstate_offset[2]:  576, xstate_sizes[2]:
 ↪256

[ .... ]
```

Using the same crash utility, you can drill even deeper into memory locations and their contents, what is being handled by every CPU core at the time of the crash and so much more. If you want to learn more about these functions, simply type `help` followed by the function name:

```
crash> help mount
```

Something similar to a man page will load onto your screen.

Summary

So, there you have it: an introduction into kernel crash debugging. This article barely scrapes the surface, but hopefully, it will provide you with a proper starting point to help diagnose kernel crashes in production, development and test environments. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior performance software engineer at Cray for its Lustre High Performance File System division. He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

DEEP DIVE

A Conversation with Kernel Developers from Intel, Red Hat and SUSE

Three kernel developers describe what it's really like to work on the kernel, how they interact with developers from other companies, some pet peeves and how to get started.

By Bryan Lunduke

Like most Linux users, I rarely touch the actual code for the Linux kernel. Sure, I've looked at it. I've even compiled the kernel myself on a handful of occasions—sometimes to try out something new or simply to say I could do it ("Linux From Scratch" is a bit of a right of passage).

But, unless you're one of the Linux kernel developers, odds are you just don't get many opportunities to truly look "under the hood".

Likewise, I think for many Linux users (even the pro users, sysadmins and developers), the wild world of kernel development is a bit of a mystery. Sure, we have the publicly available Linux Kernel Mailing List (LKML.org) that anyone is free to peruse for the latest features, discussions and (sometimes) shenanigans, but that gives only a glimpse at one aspect of being a kernel developer.

And, let's be honest, most of us simply don't have time to sift through the countless pull requests (and resulting discussions of said pull requests) that flood the LKML on a daily basis.

With that in mind, I reached out to three kernel developers—each working at some of the most prominent Linux contributing companies today—to ask them some basic questions that might provide a better idea of what being a Linux kernel developer is truly like: what their days look like and how they work with kernel developers at other companies.

Those three developers (in no particular order):

- Dave Hansen, Principal Engineer, System Software Products at Intel.
- Josh Poimboeuf, Principal Software Engineer on Red Hat Enterprise Linux.
- Jeff Mahoney, Team Lead of Kernel Engineering at SUSE Labs.

Intel, Red Hat and SUSE—three of the top contributors of code to the Linux kernel. If anyone knows what it's like being a kernel developer, it's them.

I asked all three the exact same questions. Their answers are here, completely unmodified.

Bryan Lunduke: How long have you been working with the Linux kernel? What got you into it?

Dave Hansen (Intel): My first experience for the Linux kernel was a tiny little device driver to drive the eight-character display on an IBM PS/2, probably around 20 years ago. I mentioned the project on my college resume, which eventually led to a job with IBM's Linux Technology Center in 2001. IBM is where I started doing the Linux kernel professionally.

Josh Poimboeuf (Red Hat): My first introduction to Linux happened around 2001, when I was a software engineer at IBM working on server firmware. IBM had recently embraced Linux, and I was placed on a team that was responsible for replacing legacy proprietary firmware with a new embedded PowerPC platform based on Linux.

Once I discovered Linux, I was hooked. I installed it on my laptop immediately. It was mind-boggling that all the source code was freely available, and that you could control every single bit of code that ran on your laptop. And, unbelievably, it was free.

At IBM, I started out by doing hardware bringup and Linux application development. But I was always especially fascinated by the kernel. So my curiosity gradually led me to work my way down the stack. My first real kernel experience came when I started writing device drivers in 2004.

By 2008 I was the "kernel guy" on the team, responsible for porting the kernel to our proprietary HW, and for resolving all kernel issues found in the field. That was a bit of a trial by fire, but it was a great way to learn about the entire kernel tree.

The kernel is so big that my learning process still continues to this day. That's always

been one of my favorite things about the kernel. There's always more to learn.

These days I work at Red Hat, where I do a lot more upstream work. I work on a wide variety of things: live patching, the objtool static analysis tool, the x86 unwinder, speculative CPU vulnerability mitigations and more.

Jeff Mahoney (SUSE): I've been working with the Linux kernel for 20 years. I got into it initially in college because I was interested in systems software. I happened to buy some hardware for which there was no driver, and I wrote a small one. Before working on the kernel, I was a systems admin for UNIX systems, and a coworker and I then decided to try writing a clustered filesystem ourselves. It turned out that the hardware to do that was much too expensive for us, so we ended up contributing to ReiserFS instead. That led both of us into careers working on Linux.

Bryan: What does a regular day as a Linux kernel developer look like?

Dave (Intel): My days can vary a lot. The one constant is probably email—lots and lots of email. It might be internal or external code reviews, or answering a question from another Intel team or an external customer. The best, most satisfying days are the ones where you start with a problem or a kernel crash, and have a patch posted by the end of the day.

Josh (Red Hat): It might actually be surprising to learn the variety of things a kernel developer does on a daily basis. Each day—and week and month—is different. It's often "choose your own adventure".

Obviously, one of the main things a kernel developer does is write code. Sometimes I can (mostly) disappear for a week or month (or two!) to hack away on a new feature. Those days/weeks/months are my favorite part.

But writing code is only part of it. There's also debugging, reading code, collaborating, testing, code review, code-related discussions, reading papers, research and meetings. It's good to mix things up. And you get to interact with some really smart people from

all over the world, which is really interesting.

Most communication happens through email, but many kernel developers also attend conferences, like Linux Plumbers Conference or Kernel Recipes. A lot of good discussion happens at conferences. They're also useful for putting faces to names, which makes a big difference when you're mostly interacting with people over email.

Jeff (SUSE): It's a mix of communication, coding, building and testing. Lots of email—bug reports, code review, design discussions either internally, on public mailing lists or IRC.

Bryan: With the kernel work your team does, how much of it is working with others within your own company, and how much is working with developers working on Linux at other companies? Possibly companies that compete with each other in some ways?

Dave (Intel): Because of the incredible variety in the way that our customers use Linux, work with the upstream kernel is an absolute imperative. Virtually all of that work results in work with folks in the community from other companies. There is also a lot of work that goes on behind the scenes to support the work with the community. A great comment from someone in the community might result in us going off for a week or a month to revise our work. Although we might not be sending mail on LKML on a given day, we are actively working with the community.

Josh (Red Hat): It really varies by person. Some people spend 100% of their time working upstream with the Linux community. Others spend 100% of their time internally, backporting and resolving issues in RHEL. Many of us are somewhere in the middle, dividing our time in both worlds.

Red Hat's policy is "upstream first". So any features or fixes in Red Hat's kernel have to be accepted by the upstream community first. That gives us plenty of opportunities to work with the Linux community.

Jeff (SUSE): It depends on what we're doing—bug reports tend to be handled within

SUSE, by our developers and support teams. Part of that is contractual, while part of it is practical. When we release a product, we've chosen a particular kernel version and build on top of that. Any fixes must also be against that version, and the upstream community isn't usually interested in those. Once we've created a fix, if the bug still exists in the latest release, we'll do that work in public.

Feature development happens on public mailing lists, where the participants may work at SUSE, may work for other companies, or may be doing it out of personal interest. One of the most enjoyable parts of working on Linux is that even though there are developers from hundreds of different companies who may be competing with one another, we get to collaborate as if we were on a single team. In addition to the mailing lists used to do code review and discussions, many subsystems have IRC channels where developers (and users) chat about projects and socialize.

Bryan: When you need to work with other companies (be it Intel, SUSE, Red Hat, Canonical, IBM and so on) on kernel issues—such as security vulnerabilities—how does that work? Is there an established process?

Dave (Intel): There are really two complementary processes that happen. Intel has formal company-to-company communication channels that are really great for synchronizing the business side of things. A challenge on the security front has been creating communication channels that support the coordinated disclosure process and simultaneously support normal community processes, like mailing lists. Both avenues have matured quickly and continue to evolve to help us meet the evolving security landscape.

Josh (Red Hat): When there's an embargoed security vulnerability, we do have strict processes in place for secure collaboration with other companies.

Luckily, such embargoes are rare, and they're the exception to the rule for how we normally operate. We typically work closely with developers from other companies all the time on the Linux kernel mailing list, with no special processes needed.

One good example is live kernel patching. My team at Red Hat created the kpatch technology, but at the same time, a team at SUSE created kGraft. Instead of going forward with competing approaches, we worked closely with the SUSE team at conferences and through mailing lists to create livepatch, which actually turned out to be a better technology than both kpatch and kGraft.

In fact, cross-company collaborations like that happen every day on the upstream mailing lists. It's really just business as usual. Our interactions are always focused on what's best for upstream. In the end, what's best for upstream is also what's best for the companies that rely on it. That independent company-agnostic attitude is strongly reflected in the upstream Linux culture.

Jeff (SUSE): For security vulnerabilities that aren't yet public, our security team coordinates with their counterparts with other companies. Otherwise, unless there's a compelling reason not to, the collaboration all happens on public mailing lists. There, the process is to post your code, listen and respond to review and feedback, perform the required changes, re-post, and repeat.

When the feedback is positive, the process is complete, the maintainer for the subsystem will pick it up (according to their timeline) and pull it into the git repository for their subsystem. Then the maintainer asks Linus to pull those changes into the mainline repository.

Bryan: Every software developer has a pet peeve with the projects they work on. What's your pet peeve—the thing that you really wish you could change—with Linux?

Dave (Intel): I really wish developers would focus on making reviewers' lives easier. First, communicating what you are doing, why you are doing it and why it matters is critical. Then, making sure that the code and its supporting comments are as self-explanatory as possible. Too often, we focus on making sure the code functions, then call it a day. To me, that's only half of the job.

Josh (Red Hat): If I had a magic Linux wand, I would:

1. Eliminate CPU speculation—no more Spectre/Meltdown-type bugs!

2. Get rid of the need for security embargoes—but to be clear, I believe that such embargoes are necessary in the real world.

3. More broadly diversify the Linux kernel development population. More differing perspectives can produce better ideas. I think we're already slowly moving in that direction.

Of those, #1 and #2 aren't realistic, but maybe we can achieve #3.

Jeff (SUSE): The lack of diversity in the community, especially the gender gap. Women are underrepresented in computer science fields generally, but especially so in the Linux kernel community. There has been some outreach efforts, but more needs to be done.

Bryan: The Linux kernel is, at this point, more than a quarter of a century old. In software terms, it's certainly been around a while! Do you see the need for it to be replaced any time soon? If so, with what? If not, why?

Dave (Intel): I don't really see Linux as a 25-year-old project. The Linux of 25 years ago was not the de facto OS on servers, routers or phones. There's no need to replace something that's continually changing, growing and improving as fast as Linux.

Josh (Red Hat): These days, tech trends are fickle, and most technologies have a very short lifetime. But I don't see Linux going anywhere. Its true strength is in its development model. It's not perfect, but it's still the best way to produce software at scale that I've ever seen. I wouldn't be surprised to see Linux thrive well into the next quarter century.

Jeff (SUSE): While the project is more than 25 years old, it hasn't stood still. The kernel itself is constantly evolving to meet new needs. New kinds of users

are coming to the Linux community every year. The kernel has had between 10,000 and 15,000 commits in each release for at least the past ten years. The community is still growing. I don't think it will be replaced any time soon, but it will continue to evolve.

Bryan: What would you tell folks thinking of getting into Linux kernel development?

Dave (Intel): Please do! Linux is only becoming more important to companies like Intel. It's a challenge to find folks with the technical skills to work on the kernel and the skills necessary to navigate the community. The most successful folks who join the community are the ones that have a *problem* to work on. Maybe it's some device that Linux doesn't support, or a bug that's driving you crazy on your laptop. The folks that come with patches that don't solve a clear problem generally have a tough time getting those patches accepted.

Josh (Red Hat): First, I'd say to just find a way to dive in and see if you like it.

One good way to get started is to pick a small area of the kernel you're interested in and dedicate yourself to becoming an expert on that little piece of code. Read the code until you understand it. Tweak it and see how it affects your system. Start reviewing patches related to it on the mailing list. After a while, you'll start seeing opportunities for patches, like bug fixes or code improvements.

When you do eventually post a patch, don't get overly attached to your code. Try not to take feedback personally. Our common goal is to produce the best code. It's ok to make mistakes. Put your ego aside, be humble, be respectful, and listen to feedback with an open mind and try to learn from it. That's how the code gets better. It also helps you earn respect from others in the community.

Kernel development can take a lot of patience, humility and persistence. It's not uncommon for code to be thrown away or rewritten several times. The process

can seem inefficient at times. But in my experience, the end result is always better than anything proprietary development can produce.

The kernel codebase is huge, so diving into code you've never seen will be a common occurrence. Whenever you have a question about how something works, the answer is always in the code somewhere. Get familiar with cscope. For vim users, I'd recommend the vim cscope plugin.

Also, work on your written communication skills, as most of your non-coding time will be spent in email. And, of course, learning to make your code easily readable by others is also very important.

Finally, finding a mentor (or mentors) can be valuable. I never had an official mentor per se, but I've been lucky enough to have had many more experienced people guide me through the years.

Jeff (SUSE): It can be a lot of fun, but it takes some effort to get started. Start with something you're interested in, find something small to fix, and post your work. Read about and understand the process. Listen and respond to feedback. Experienced developers are usually willing to spend some time helping new developers if they're willing to listen. ■

Bryan Lunduke is a former Software Tester, former Programmer, former VP of Technology, former Linux Marketing Guy (tm), former openSUSE Board Member... and current Deputy Editor of *Linux Journal* as well as host of the popular *Lunduke Show*. More details: **http://lunduke.com**.

Send comments or feedback
via https://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Using Machine Learning to Optimize Linux Networking

The Linux networking stack can benefit from "inferences" due to machine learning, which may be used in "smart" applications.

By Damian Valles and Stan McClellan

Machine Learning (ML) is driving the exploration of vast volumes of data. With the right training data, ML can outperform many traditional forms of automation and analysis in diverse industries and applications. A critical advantage of this approach is that ML can cut through human biases and protocols established over decades in some cases. ML even can be used on Linux kernel-level data streams to optimize networked activities—and to enable the system to "understand" its environment. Here, we use ML on data that is constantly generated by the Linux networking stack to provide an additional, rudimentary form of "intelligence" about networked systems that are nearby. We approach this problem by allowing ML algorithms to work on the byproducts of the Stream Control Transmission Protocol (SCTP).

SCTP is a relatively new transport protocol for IP networks. Defined originally in RFC 4960, it provides many reliability benefits, such as multi-homing, multi-streaming and path selection, which are useful in control-plane or signaling applications. SCTP is used in place of conventional transport protocols (such as TCP and UDP) in telecommunications, Smart Grid, Internet-of-Things (IoT) and Smart Cities applications, among others. We chose to use SCTP for this ML experiment because it has many useful applications and characteristics. However,

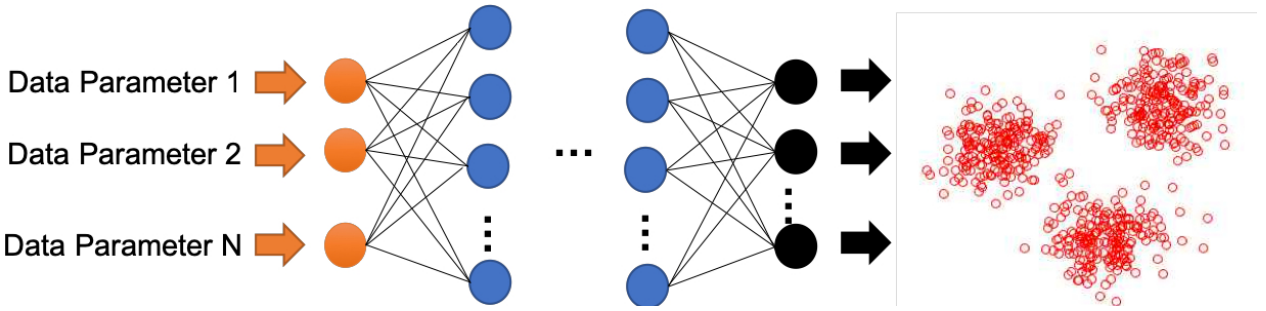


Figure 1. Deep learning neural network: orange nodes are the input layer, blue nodes are hidden layers, and black nodes are the output layer. The red circles on the far right are outputs that have been "clustered" by the network.

the use of ML approaches in different types of kernel-level data streams also may be useful for classifying congestion patterns, forecasting data movement and providing new application-specific "intelligence".

But, before going further in the use of ML techniques inside the Linux kernel, we should discuss some background concepts. The most popular form of ML is Deep Learning (DL) Neural Network (NN) algorithms. An NN model consists of an input neural layer, hidden layers and output layer. Figure 1 shows these layers as orange, blue and black nodes, respectively. The number of inputs depends on the number of parameters that are considered to develop the model. The hidden neural layers can be configured in many forms to perform different statistical analysis using weight factors and activation functions. The number of outputs in the last neural layer provides the weighted outcomes of the model.

NN models are trained using approaches known as supervised, unsupervised and reinforcement learning, which are described briefly in the following sections.

Supervised training uses data with labels that tell the network which outputs should be produced by specific input parameters. The training process reconfigures weighting values within the hidden and output layers to ensure these outcomes. The more data presented to the network during training, the more accurate outcomes appear when inputs are not in the training data.

The trained NN model is then validated using input data that is similar to the training data but without labels or desired outputs. Validation outputs should be similar to training outputs. The training-validation cycle helps a designer understand the percentage of accuracy the network has reached after being trained. If the accuracy is too low for the design, further training is required, and different NN design parameters are tuned. The training and validation phases must be tuned iteratively until the desired or required accuracy appears.

The final step is the test phase of the NN model. At this point, the model is fed input data not presented during training or validation phases. The data set used in the test phase is often a small subset of the overall data set used for the NN model design. The test phase results are known as the real values of accuracy.

Unsupervised training uses training data that is not labeled. The network "learns" autonomously by adjusting its weights based on patterns detected in the input data. Unsupervised designs typically analyze input data that is unknown or unstructured from the designer's perspective. As a result, this approach takes longer for the NN model to be trained and requires more data if high accuracy is required.

Reinforcement learning is becoming the most robust ML training approach to manage complex problems in many industries and research fields. This approach trains the model through a reward system. Random scenarios circulate to the input, and a reward is fed to the model when reaching the desired outcome. This approach requires much computational time to train the NN model due to the learning curve. However, reinforced models have reached a high competency for solving problems and out-performing human experts in many applications. One of the most effective reinforcement learning examples was mastered by Google's AlphaGo team to win the ancient game *Go*. The NN model was able to beat the best *Go* players in the world.

To incorporate ML techniques into Linux networking processes, we use Round-Trip Times (RTT) and Retransmission Timeouts (RTO) values as input values

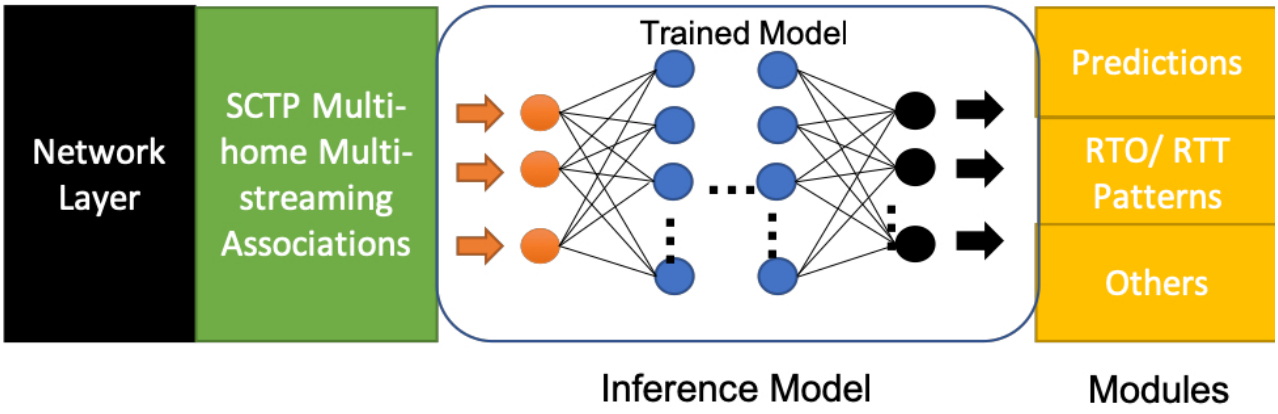


Figure 2. Inference model implementation: RTT/RTO values from SCTP are fed to an inference model to produce smarter congestion handling.

for a layered NN model trained via unsupervised and supervised learning. In this approach, RTT/RTO values from a network interaction based on SCTP are collected, then used to train the NN model. Using this data, the trained model produces outcomes that create new insights (or "inferences") into the networking context of the system. The objective is to see if the ML system can differentiate between various known network scenarios, or if the ML system can produce new information about network activities. In our results, the RTT and RTO values already exist inside the Linux kernel, and the insights/outcomes that result from the ML process are unique and useful.

The approach developed to integrate an NN model to devices is known as an inference model. An inference model is an optimized NN model code engine that can run on a computer device. A useful inference model generator comes from NVIDIA's TensorRT Programmable Inference Accelerator application. This way, inference models through TensorRT then can be imported to the Linux kernel for networking modules, scheduling of processes, priority calculations or other kernel functionalities that can provide a smarter execution flow for application-specific implementations.

The goal here is to obtain a smarter network flow using the SCTP protocol and an

ML-based evaluation of related kernel-level data. The results of our experiments clearly show that it's possible to optimize network activities by using an inference model to digest the current state and select the right processing or subsequent state. Figure 2 shows the inference model implementation situated between reading metrics from the SCTP module in the Linux kernel and the selection of outcomes. The inference model reads input parameters from the current state of network activities, then processes the data to determine an outcome that better "understands" the state of the network. Given that the implementation may be application-specific, the inference model also may be trained to have different outcomes biased to the application.

We created four different scenarios to test several ML implementations in the Linux kernel. These scenarios are detailed below, and they represent a wireless device moving away from or toward a wireless access point with a few variations, including the access point being in signal over-saturation. In our experiments, 2,500 samples were recorded for each scenario and left unlabeled. Each data set collected was normalized against its maximum value. By normalizing the data, patterns become more pronounced, which improves the ML analysis. Here are the scenarios:

1. A device moving away/toward, 1 meter to 25 meters away.

2. A device moving 15 meters away/toward, rounding a corner and continuing 10 meters further.

3. A device moving 25 meters away/toward with a wall between it and the wireless source.

4. A device moving 25 meters away/toward with two walls between it and the wireless source.

Congestion due to other devices is a common phenomenon in a wireless network. To model this issue, we included a collection of cross traffic in the

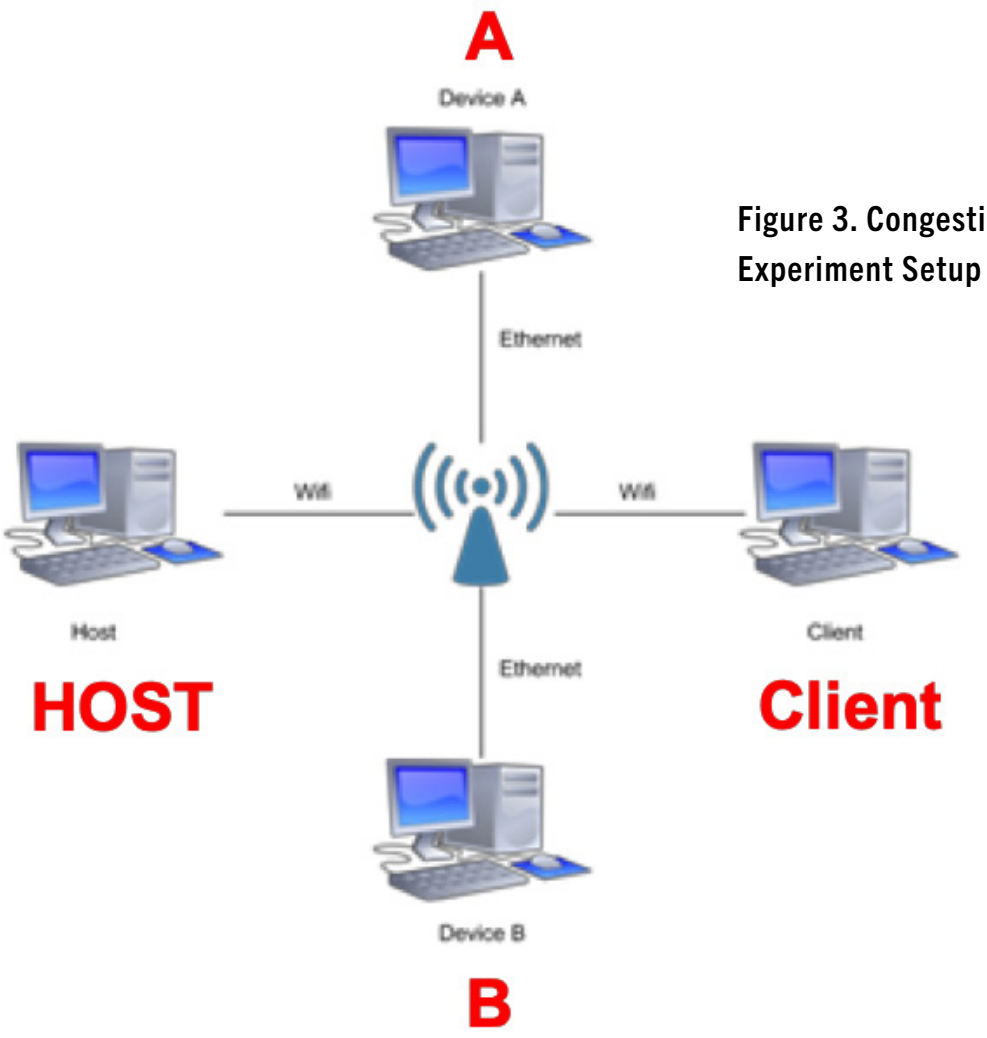


Figure 3. Congestion Experiment Setup

network to evoke congestion. Figure 3 shows the system designations, where A/B systems provided cross traffic and are connected via wired Ethernet, and Host/Client systems provided test data and are connected via wireless. In this scenario, 5,000 15-second samples were collected from the Client node as it sent SCTP-Test packets to the Host node, while A-to-B traffic consisted of 1MB, 512KB, 256KB, 128KB and 1KB chunk size FTP traffic.

To create additional cross-traffic variation, we also used the VLC media player to stream a file over UDP from A-to-B. In this scenario, we collected 5,000 15-second samples from the Client device as it sent SCTP-Test packets to the

Host device. Since the bitrate of the UDP stream changed continuously due to compression algorithms, further adjustments in chunk sizes were unnecessary.

To classify the data obtained from our network scenarios, we used four ML classification algorithms that were trained using unsupervised or supervised learning techniques. These ML techniques are briefly described below.

K-Means Clustering:

- K-Means is an algorithm for clustering data. Though computationally expensive, K-Means is good at creating clusters with very high learning rates.

- Since K-Means is an unsupervised algorithm, it allows for independent interpretation of the results.

Support Vector Machine (SVM):

- SVMs are a kind of supervised learning method that can be used for clustering. SVMs use a subset of the training data in the decision function, making it memory-efficient.

- Using an SVM with the Radial Basis Function (RBF) kernel with a moderate, its C-parameter proved computationally expensive but had the best training rates.

Decision Trees (DT):

- DTs are a supervised learning method for classification. In creating simple rules from labeled data features, it attempts to predict the values of the target variable.

- DTs have prediction costs logarithmic to the number of training sets. In fact, DTs are exceptional at using incomplete data sets and training models with low numbers of sets and still yielding high similarity quotients with statistical methods like K-means.

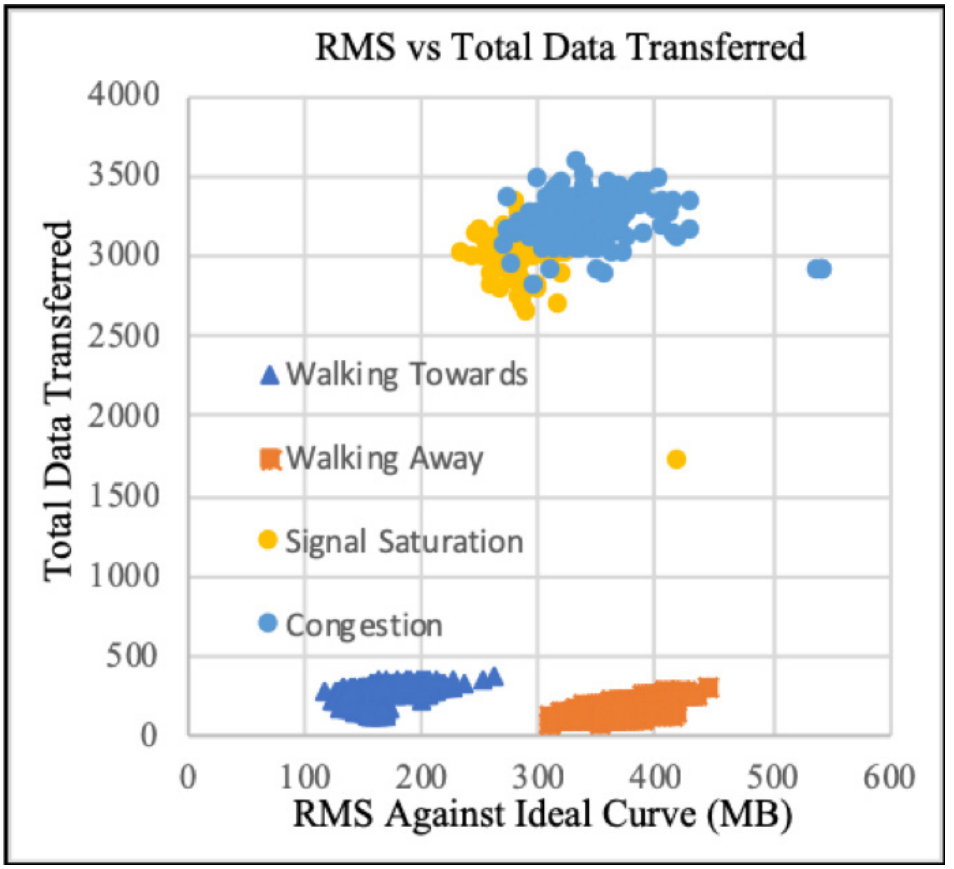


Figure 4. ML classification outcomes of Linux kernel network data using the K-Means algorithm. The classified data clearly indicates four different outcomes for external networked systems.

Nearest Neighbor Classifier (NCC):

- NCC is a supervised NN model that uses centroids to define boundaries when classifying data sets. NCCs also are good at classifying sets where probability distributions are unknown and can respond to changes quickly.

- Since network delay patterns are likely to change as devices and users are introduced or removed from the network, an NCC may be better at adapting to those changes.

As an example of the outcomes of our testing, Figure 4 shows the results of classifying the normalized network data using K-Means clustering. Note from Figure 4 that four differentiated outcomes are clearly present: 1) systems

"moving toward" or 2) "moving away" from each other, as well as systems experiencing 3) "signal saturation" and (4) "network congestion". Note that regardless of the test scenario, the ML algorithm can discern valuable information about the networked systems. Although additional analysis can produce more information, these outcomes may be quite useful in certain applications.

The ideal outcome or classification for the different network testing scenarios is a clear separation between the four groups, as shown in Figure 4. Two of the cases are clearly and individually separated from the group. However, in the cases of congestion and signal over-saturation, the classification overlaps without a clear separation. This lowers the accuracy of the ML model. To improve the classification, we would need to include another parameter in the analysis that helps to discriminate between congestion and saturation.

In all cases, the ML techniques analyzed data that is consistently produced internal to the Linux kernel to gain new insights about the external network context. However, since these algorithms are operating on kernel-level data, the issue of efficiency is also important.

Figure 5 summarizes the accuracy and efficiency of classifying the network data using ML algorithms. From Figure 5, it's clear that the SVM approach is most efficient: it achieves the highest accuracy using the least number of training samples. This means that the SVM classifier can quickly distinguish differences in the kernel data and can produce useful outcomes efficiently. It is also an excellent practice to realize a large number of examples to see the convergence of accuracy for each classifier. However, NN models that reach 100% accuracy may produce large failure rates due to overfitting.

Accurate and fast classification of network data using ML techniques may result in Linux systems that can autonomously react based on external network conditions. These reactions may include modifying kernel data, selecting alternate retransmission or transport protocols, or adjusting other internal

Number of Training Samples	Accuracy of K-Means	Accuracy of SVM	Accuracy of Decision Tree	Accuracy of Nearest Centroid
150	90.67%	96.55%	96.67%	68.67%
500	92.11%	98.96%	97.81%	79.33%
1,500	92.82%	99.99%	98.13%	86.23%
15,000	92.82%	99.99%	98.21%	86.67%

Figure 5. Accuracy and speed of classification algorithms when using normalized data from different networking scenarios.

parameters based on external context. This form of "inference" is a fundamental problem in systems communicating via a network. In our experiments, network data was collected directly from the existing kernel processes and used to train the four ML classifiers to produce interesting and useful inferences.

Our results suggest that the SVM approach may be a promising ML technique for inferring results from kernel networking data. The SVM approach reduces the number of nodes per layer, adjusts precision of the data without losing accuracy and reduces the latency of computation. The output of the inference layer provides the networking stack with additional information to handle different scenarios. The inference development represents a form of customizing network communication between endpoints for a variety of applications. Other network conditions may benefit from inference models that utilize different data produced by the networking stack.

As ML techniques are included in new designs and applications, they have become a valuable tool and part of the implementation process for smarter networks. Concepts related to "machine learning" and "artificial intelligence" even may be implemented inside the Linux kernel to improve networking performance. ■

Resources

- "Introduction to Stream Control Transmission Protocol" by Jan Newmarch, *LJ*, September 2007
- Silver, D., Schrittwieser, J., et al., "Mastering the game of Go without human knowledge", *Nature* 550: 354–359, Macmillan Publishers Limited, DOI: 10.1038/nature24270
- NVIDIA TensorRT Programmable Interface Accelerator
- "Introduction to K-means Clustering" by Andrea Trevino on December 6, 2016
- Towards Data Science: Support Vector Machine—Introduction to Machine Learning Algorithms by Rohith Gandhi on June 7, 2018
- Decision Trees (DTs): A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of the use of decision tree algorithms in machine learning", 2011 IEEE Control and System Graduate Research Colloquium, Shah Alam, 2011, pp. 37–42
- Nearest Centroid Classifier (NCC): V. Praveen, K. Kousalya and K. R. Prasanna Kumar, "A nearest centroid classifier-based clustering algorithm for solving vehicle routing problem," 2016 2nd International Conference on Advances in Electrical, Electronics, Information, Communication, and Bioinformatics (AEEICB), Chennai, 2016, pp. 414–419.

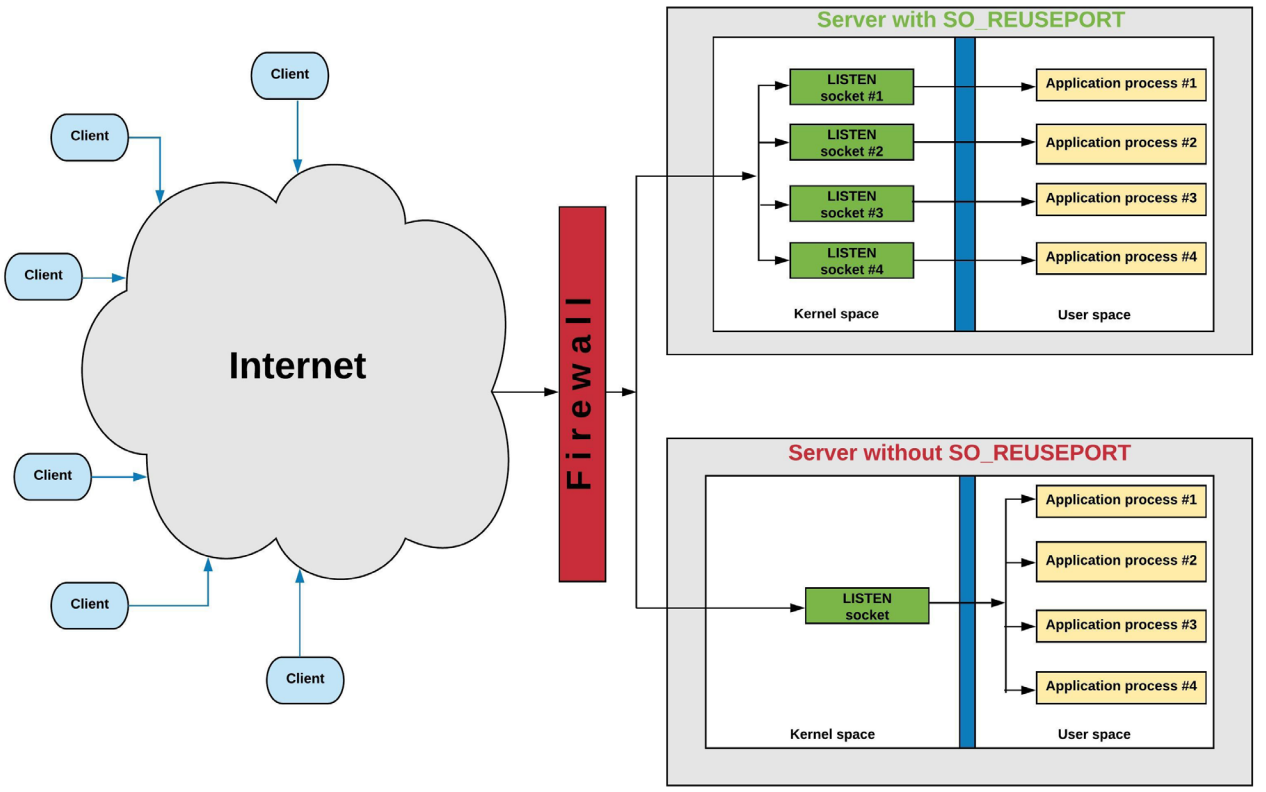
Damian Valles is a second-year Assistant Professor in the Ingram School of Engineering at Texas State University. His goal is not to partially tear another Achilles Heel anytime soon while staying active. Damian welcomes your comments at dvalles@txstate.edu or Twitter: @VallesDamian.

Stan McClellan has been an avid Linux user and network experimenter for many years. One of his interests is using Linux to help Damian avoid another athletic injury. He can be reached at s.mcclellan@ieee.org.

Linux TCP SO_REUSEPORT: Usage and Implementation

Improve your server performance using a relatively new feature of the Linux networking stack: the SO_REUSEPORT socket option.

By Krishna Kumar



HAProxy and NGINX are some of the few applications that use the TCP `SO_REUSEPORT` socket option of the Linux networking stack. This option, initially introduced in 4.4 BSD, is used to implement high-performance servers that help better utilize today's large multicore systems. The first few sections of this article explain some essential concepts of TCP/IP sockets, and the remaining sections use that knowledge to describe the rationale, usage and implementation of the `SO_REUSEPORT` socket option.

TCP Connection Basics

A TCP connection is defined by a unique 5-tuple:

> [*Protocol, Source IP Address, Source Port, Destination IP Address, Destination Port*]

Individual tuple elements are specified in different ways by clients and servers. Let's take a look at how each tuple element is initialized.

Client Application

Protocol: this field is initialized when the socket is created based on parameters provided by the application. The protocol is always TCP for the purposes of this article. For example:

```
socket(AF_INET, SOCK_STREAM, 0);  /* create a TCP socket */
```

Source IP address and port: these are usually set by the kernel when the application calls `connect()` without a prior invocation to `bind()`. The kernel picks a suitable IP address for communicating with the destination server and a source port from the ephemeral port range (`sysctl net.ipv4.ip_local_port_range`).

Destination IP address and port: these are set by the application by invoking `connect()`. For example:

```
server.sin_family  = AF_INET;
```

```
server.sin_port    = htons(SERVER_PORT);
bcopy(server_ent->h_addr, &server.sin_addr.s_addr,
      server_ent->h_length);
/* Connect to server, and set the socket's destination IP
 * address and port# based on above parameters. Also, request
 * the kernel to automatically set the Source IP and port# if
 * the application did not call bind() prior to connect().
 */
connect(fd, (struct sockaddr *)&server, sizeof server);
```

Server Application

Protocol: initialized in the same way as described for a client application.

Source IP address and port: set by the application when it invokes `bind()`, for example:

```
srv_addr.sin_family        = AF_INET;
srv_addr.sin_addr.s_addr   = INADDR_ANY;
srv_addr.sin_port          = htons(SERVER_PORT);
bind(fd, &srv_addr, sizeof srv_addr);
```

Destination IP address and port: a client connects to a server by completing the TCP three-way handshake. The server's TCP/IP stack creates a new socket to track the client connection and sets its Source IP:Port and Destination IP:Port from the incoming client connection parameters. The new socket is transitioned to the `ESTABLISHED` state, while the server's `LISTEN` socket is left unmodified. At this time, the server application's call to `accept()` on the `LISTEN` socket returns with a reference to the newly `ESTABLISHED` socket. See the listing at the end of this article for an example implementation of client and server applications.

TIME-WAIT Sockets

A TIME-WAIT socket is created when an application closes its end of a TCP connection first. This results in the initiation of a TCP four-way handshake, during which the socket state changes from `ESTABLISHED` to `FIN-WAIT1` to `FIN-WAIT2`

to `TIME-WAIT`, before the socket is closed. The `TIME-WAIT` state is a lingering state for protocol reasons. An application can instruct the TCP/IP stack not to linger a connection by sending a TCP RST packet. In doing so, the connection is terminated instantly without going through the TCP four-way handshake. The following code fragment implements the reset of a connection by specifying a socket linger time of zero seconds:

```
const struct linger opt = { .l_onoff = 1, .l_linger = 0 };

setsockopt(fd, SOL_SOCKET, SO_LINGER, &opt, sizeof opt);
close(fd);
```

Understanding the Different States of a Server Socket

A server typically executes the following system calls at start up:

1) Create a socket:

```
server_fd = socket(...);
```

2) Bind to a well known IP address and port number:

```
ret = bind(server_fd, ...);
```

3) Mark the socket as passive by changing its state to `LISTEN`:

```
ret = listen(server_fd, ...);
```

4) Wait for a client to connect and get a reference file descriptor:

```
client_fd = accept(server_fd, ...);
```

Any new socket, created via `socket()` or `accept()` system calls, is tracked in the kernel using a "struct sock" structure. In the code fragment above, a socket

is created in step #1 and given a well known address in step #2. This socket is transitioned to the `LISTEN` state in step #3. Step #4 calls `accept()`, which blocks until a client connects to this IP:port. After the client completes the TCP three-way handshake, the kernel creates a second socket and returns a reference to this socket. The state of the new socket is set to `ESTABLISHED`, while the `server_fd` socket remains in a `LISTEN` state.

The SO_REUSEADDR Socket Option

Let's look at two use cases to better understand the `SO_REUSEADDR` option for TCP sockets.

Use case #1: a server application restarts in two steps, an exit followed by a start up. During the exit, the server's `LISTEN` socket is closed immediately. Two situations can arise due to the presence of existing connections to the server:

1. All established connections that were being handled by this dying server process are closed, and those sockets transition to the `TIME-WAIT` state.

2. All established connections that were handed off to a child process continue to remain in the `ESTABLISHED` state.

When the server is subsequently started up, its attempt to bind to its `LISTEN` port fails with `EADDRINUSE`, because some sockets on the system are already bound to this IP:port combination (for example, a socket in either the `TIME-WAIT` or `ESTABLISHED` state). Here's a demonstration of this problem:

```
# Server is listening on port #45000.
$ ss -tan | grep :45000
LISTEN   0     1      10.20.1.1:45000      *:*

# A client connects to the server using its source
# port 54762. A new socket is created and is seen
# in the ESTABLISHED state, along with the
```

```
# earlier LISTEN socket.
$ ss -tan | grep :45000
LISTEN     0     1    10.20.1.1:45000      *:*
ESTAB      0     0    10.20.1.1:45000      10.20.1.100:54762

# Kill the server application.
$ pkill -9 my_server

# Restart the server application.
$ ./my_server 45000
bind: Address already in use

# Find out why
$ ss -tan | grep :45000
TIME-WAIT  0     0    10.20.1.1:45000      10.20.1.100:54762
```

This listing shows that the earlier `ESTABLISHED` socket is the same one that is now seen in the `TIME-WAIT` state. The presence of this socket bound to the local address—10.20.1.1:45000—prevented the server from being able to subsequently `bind()` to the same IP:port combination for its `LISTEN` socket.

Use case #2: if two processes attempt to `bind()` to the same IP:port combination, the process that executes `bind()` first succeeds, while the latter fails with `EADDRINUSE`. Another instance of this use case involves an application binding to a specific IP:port (for example, 192.168.100.1:80) and another application attempting to bind to the wildcard IP address with the same port number (for example, 0.0.0.0:80), or vice versa. The latter `bind()` invocation fails, as it attempts to bind to all addresses with the same port number that was used by the first process. If both processes set the `SO_REUSEADDR` option on their sockets, both sockets can be bound successfully. However, note this caveat: if the first process calls `bind()` and `listen()`, the second process still would be unable to `bind()` successfully, since the first socket is in the `LISTEN` state.

Hence, this use case is usually meant for clients that want to bind to a specific IP:port before connecting to different services.

How does `SO_REUSEADDR` help solve this problem? When the server is restarted and invokes `bind()` on a socket with `SO_REUSEADDR` set, the kernel ignores all non-`LISTEN` sockets bound to the same IP:port combination. The *UNIX Network Programming* book describes this feature as: "`SO_REUSEADDR` allows a listening server to start and bind its well-known port, even if previously established connections exist that use this port as their local port".

However, we need the `SO_REUSEPORT` option to allow two or more processes to invoke `listen()` on the same port successfully. I describe this option in more detail in the remaining sections.

The SO_REUSEPORT Socket Option

While `SO_REUSEADDR` allows sockets to `bind()` to the same IP:port combination when existing `ESTABLISHED` or `TIME-WAIT` sockets may be present, `SO_REUSEPORT` allows binding to the same IP:port when existing `LISTEN` sockets also may be present. The kernel ignores all sockets, including sockets in the `LISTEN` state, when an application invokes `bind()` or `listen()` on a socket with `SO_REUSEPORT` enabled. This permits a server process to be invoked multiple times, allowing many processes to listen for connections. The next section examines the kernel implementation `SO_REUSEPORT`.

How Are Connections Distributed among Multiple Listeners?

When multiple sockets are in the `LISTEN` state, how does the kernel decide which socket—and, thus, which application process—receives an incoming connection? Is this determined using a round-robin, least-connection, random or some other method? Let's take a deeper look into the TCP/IP code to understand how socket selection is performed.

Notes:

- Data structures and code snippets in this section are heavily simplified for the sake of clarity—removing some structure elements, function arguments, variables and unnecessary code—but without losing correctness. Some parts of the listing are also in pseudo-code for better ease of understanding.
- `sk` represents a kernel socket data structure of type "struct sock".
- `skb`, or the socket buffer, represents a network packet of type "struct sk_buff".
- `src_addr`, `src_port` and `dst_addr`, `dst_port` refers to source IP:port and destination IP:port, respectively.
- Readers can correlate the code snippets with the actual source code, if desired.

As an incoming packet (`skb`) moves up the TCP/IP stack, the IP subsystem calls into the TCP packet receive handler, `tcp_v4_rcv()`, providing the `skb` as argument. `tcp_v4_rcv()` attempts to locate a socket associated with this `skb`:

```
sk = __inet_lookup_skb(&tcp_hashinfo, skb, src_port, dst_port);
```

`tcp_hashinfo` is a global variable of type `struct inet_hashinfo`, containing, among others, two hash tables of `ESTABLISHED` and `LISTEN` sockets, respectively. The `LISTEN` hash table is sized to 32 buckets, as shown below:

```
#define LHTABLE_SIZE 32 /* Yes, this really is all you need */
struct inet_hashinfo {
   /* Hash table for fully established sockets */
   struct inet_ehash_bucket *ehash;
   /* Hash table for LISTEN sockets */
   struct inet_listen_hashbucket listening_hash[LHTABLE_SIZE];
};
```

```
struct inet_hashinfo tcp_hashinfo;
```

`__inet_lookup_skb()` extracts the source and destination IP addresses from the incoming `skb` and passes these along with the source and destination ports to `__inet_lookup()` to find the associated `ESTABLISHED` or `LISTEN` socket, as shown here:

```
struct sock *__inet_lookup_skb(tcp_hashinfo, skb,
 ↪src_port, dst_port)
{
   /* Get the IPv4 header to know
    * the source and destination IPs */
   const struct iphdr *iph = ip_hdr(skb);

   /*
    * Look up the incoming skb in tcp_hashinfo using the
    * [ Source-IP:Port, Destination-IP:Port ] tuple.
    */
   return __inet_lookup(tcp_hashinfo, skb, iph->saddr,
           ↪src_port, iph->daddr, dst_port);
}
```

`__inet_lookup()` looks in `tcp_hashinfo->ehash` for an already established socket matching the client four-tuple parameters. In the absence of an established socket, it looks in `tcp_hashinfo->listening_hash` for a `LISTEN` socket:

```
struct sock *__inet_lookup(tcp_hashinfo, skb, src_addr,
               ↪src_port, dst_addr, dst_port)
{
   /* Convert dest_port# from network to host byte order */
   u16 hnum = ntohs(dst_port);
```

```
   /* First look for an established socket ... */
   sk = __inet_lookup_established(tcp_hashinfo, src_addr,
            ↪src_port, dst_addr, hnum);
   if (sk)
      return sk;

   /* failing which, look for a LISTEN socket */
   return __inet_lookup_listener(tcp_hashinfo, skb, src_addr,
                                 src_port, dst_addr, hnum);
}
```

The `__inet_lookup_listener()` function implements the selection of a `LISTEN` socket:

```
struct sock *__inet_lookup_listener(tcp_hashinfo, skb,
                 ↪src_addr, src_port, dst_addr, dst_port)
{
   /*
    * Use the destination port# to calculate a hash table
    * slot# of the listen socket. inet_lhashfn() returns
    * a number between 0
    * and LHTABLE_SIZE-1 (both inclusive).
    */
   unsigned int hash = inet_lhashfn(dst_port);

   /* Use this slot# to index the global LISTEN hash table */
   struct inet_listen_hashbucket *ilb =
                     ↪tcp_hashinfo->listening_hash[hash];
   /* Track best matching LISTEN socket
    * so far and its "score" */
   struct sock *result = NULL, *sk;
   int hi_score = 0;
```

```
for each socket, 'sk', in the selected hash bucket, 'ilb' {
   /*
    * Calculate the "score" of this LISTEN socket (sk)
    * against the incoming skb. Score is computed on
    * some parameters, such as exact destination port#,
    * destination IP address exact match (as against
    * matching INADDR_ANY, for example),
    * with each criteria getting a different weight.
    */
   score = compute_score(sk, dst_port, dst_addr);
   if (score > hi_score) {
      /* Highest score - best matched socket till now */
      if (sk->sk_reuseport) {
         /*
          * sk has SO_REUSEPORT feature enabled. Call
          * inet_ehashfn() with dest_addr, dest_port,
          * src_addr and src_port to compute a
          * 2nd hash, phash.
          */
         phash = inet_ehashfn(dst_addr, dst_port,
                              src_addr, src_port);

         /* Select a socket from sk's SO_REUSEPORT group
          * using 'phash'.
          */
         result = reuseport_select_sock(sk, phash);
         if (result)
            return result;
      }

      /* Update new best socket and its score */
      result = sk;
      hi_score = score;
```

```
      }
   }

   return result;
}
```

Selecting a socket from the `SO_REUSEPORT` group is done with `reuseport_select_sock()`:

```
struct sock *reuseport_select_sock(struct sock *sk,
                                   unsigned int phash)
{
   /* Get control block of sockets
    * in this SO_REUSEPORT group */
   struct sock_reuseport *reuse = sk->sk_reuseport_cb;

   /* Get count of sockets in the group */
   int num_socks = reuse->num_socks;

   /* Calculate value between 0 and 'num_socks-1'
    * (both inclusive) */
   unsigned int index = reciprocal_scale(phash, num_socks);

   /* Index into the SO_REUSEPORT group using this index */
   return reuse->socks[index];
}
```

Let's step back a little to understand how this works. When the first process invoked `listen()` on a socket with `SO_REUSEPORT` enabled, a pointer in its "struct sock" structure, `sk_reuseport_cb`, is allocated. This structure is defined as:

```
struct sock_reuseport {
  u16      max_socks; /* Allocated size of socks[] array */
```

```
  u16      num_socks; /* #Elements in socks[] */
  struct sock *socks[0]; /* All sockets added to this group */
};
```

The last element of this structure is a "flexible array member". The entire structure is allocated such that the `socks[]` array has 128 elements of type `struct sock *`. Note that as the number of listeners increases beyond 128, this structure is reallocated such that the `socks[]` array size is doubled.

The first socket, `sk1`, that invoked `listen()`, is saved in the first slot of its own `socks[]` array, for example:

```
sk1->sk_reuseport_cb->socks[0] = sk1;
```

When `listen()` is subsequently invoked on other sockets (`sk2`, ...) bound to the

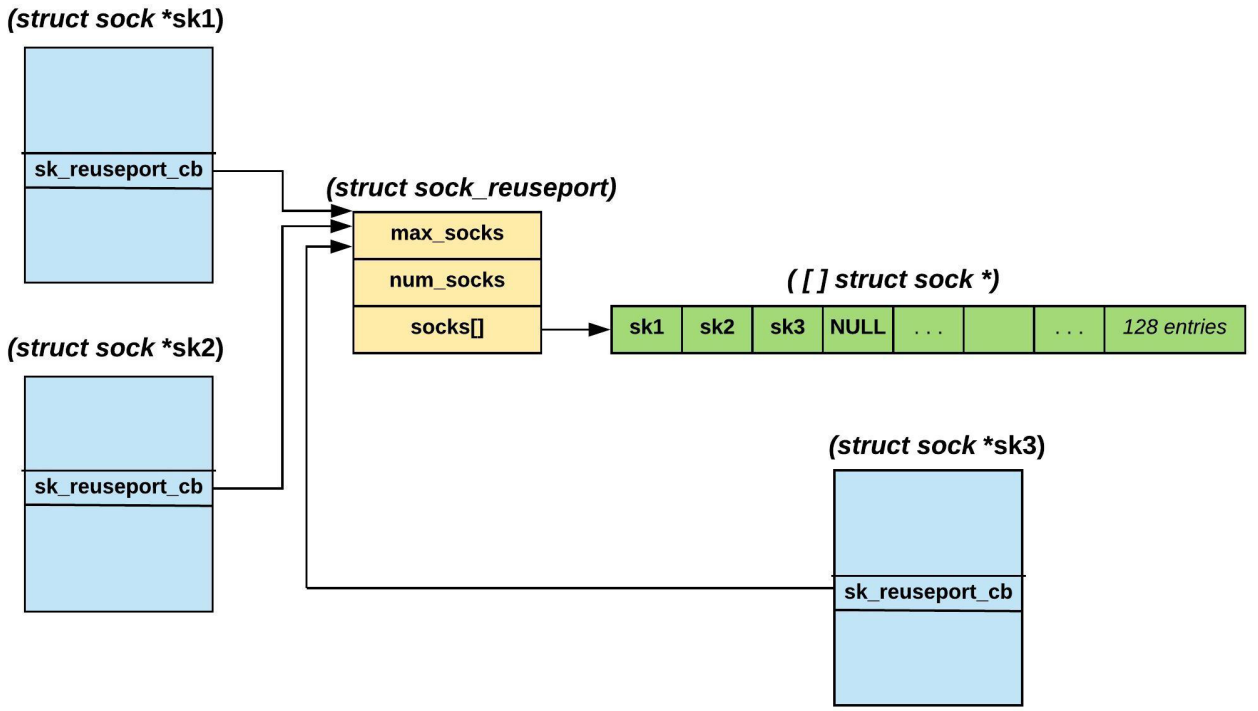


Figure 1. Representation of the `SO_REUSEPORT` Group of `LISTEN` Sockets

same IP:port, two operations are performed:

1. The address of the new socket (`sk2`, ...) is appended to the `sk_reuseport_cb->socks[]` of the first socket (`sk1`).

2. The new socket's `sk_reuseport_cb` pointer is made to point to the first socket's `sk_reuseport_cb` pointer. This ensures that all `LISTEN` sockets of the same group reference the same `sk_reuseport_cb` pointer.

Figure 1 shows the result of these two steps.

In Figure 1, `sk1` is the first `LISTEN` socket, and `sk2` and `sk3` are sockets that invoked `listen()` subsequently. The two steps described above are performed in the following code snippet and executed via the `listen()` call chain:

```
static int inet_reuseport_add_sock(struct sock *new_sk)
{
   /*
    * First check if another identical LISTEN socket, prev_sk,
    * exists. ... Then do the following:
    */
   if (prev_sk) {
      /*
       * Not the first listener - do the following:
       * - Grow prev_sk->sk_reuseport_cb structure if required.
       * - Save new_sk socket pointer in prev_sk's socks[].
       *   prev_sk->sk_reuseport_cb->socks[num_socks] = new_sk;
       * - prev_sk->sk_reuseport_cb->num_socks++;
       * - Pointer assignment of the control block:
       *   new_sk->sk_reuseport_cb = prev_sk->sk_reuseport_cb;
       */
      return reuseport_add_sock(new_sk, prev_sk);
   }
```

```
    /*
     * First listener - do the following:
     * - allocate new_sk->sk_reuseport_cb to contain 128 socks[]
     * - new_sk->sk_reuseport_cb->max_socks = 128;
     * - new_sk->sk_reuseport_cb->socks[0] = new_sk;
     * - new_sk->sk_reuseport_cb->numsocks = 1;
     */
    return reuseport_alloc(new_sk);
}
```

Now let's go back to `reuseport_select_sock()` to see how a `LISTEN` socket is selected. The `socks[]` array is indexed via a call to `reciprocal_scale()` as follows:

```
unsigned int index = reciprocal_scale(phash, num_socks);
return reuse->socks[index];
```

reciprocal_scale() is an optimized function that implements a pseudo-modulo operation using multiply and shift operations.

As shown earlier, `phash` was calculated in `__inet_lookup_listener()`:

```
phash = inet_ehashfn(dst_addr, dst_port, src_addr, src_port);
```

And, `num_socks` is the number of sockets in the `socks[]` array. The function `reciprocal_scale(phash, num_socks)` calculates an index, `0 <= index < num_socks`. This index is used to retrieve a socket from the `SO_REUSEPORT` socket group.

Hence, you can see that the kernel selects a socket by hashing the client IP:port and server IP:port values. This method provides a good distribution of connections among different `LISTEN` sockets.

SO_REUSEPORT in Action

Let's look at the effect of `SO_REUSEPORT` on the command line through two tests.

1) An application opens a socket for listen and creates two processes.

Application code path: `socket(); bind(); listen(); fork();`

```
$ ss -tlnpe | grep :45000
LISTEN     0      128    *:45000      *:* users:(("my_server",
↪pid=3020,fd=3),("my_server",pid=3019,fd=3))
 ↪ino:3854904087 sk:37d5a0
```

The string `ino:3854904087 sk:37d5a0` describes a single kernel socket.

2) An application creates two processes, and each creates a `LISTEN` socket after setting `SO_REUSEPORT`.

Application code path: `fork(); socket(); setsockopt(SO_REUSEPORT); bind(); listen();`

```
$ ss -tlnpe | grep :45000
LISTEN     0      128    *:45000      *:*  users:(("my_server",
↪pid=1975,fd=3)) ino:3854935788 sk:37d59c
LISTEN     0      128    *:45000      *:*  users:(("my_server",
↪pid=1974,fd=3)) ino:3854935786 sk:37d59d
```

Now you see two different kernel sockets—notice the different inode numbers.

Applications using multiple processes to accept connections on a single `LISTEN` socket may experience significant performance issues, since each process contends for the same socket lock in `accept()`, as shown in the following simplified pseudo-code:

```
struct sock *inet_csk_accept(struct sock *sk)
```

```
{
   struct sock *newsk = NULL;   /* client socket */

   /* We need to make sure that this socket is listening, and
    * that it has something pending.
    */
   lock_sock(sk);
   if (sk->sk_state == TCP_LISTEN)
      if ("there are completed connections waiting
            ↪to be accepted")
         newsk = get_first_connection(sk);
   release_sock(sk);

   return newsk;
}
```

Both `lock_sock()` and `release_sock()` internally acquires and releases a spinlock embedded in `sk`. (See Figure 3 later in this article to observe the overhead due to the spinlock contention.)

Benchmarking SO_REUSEPORT

The following setup is used to measure `SO_REUSEPORT` performance:

1. Kernel version: 4.17.13.

2. Client and server systems both have 48 hyper-threaded cores and are connected to each other using a 40g NIC over a switch.

3. Server is started in one of two ways: create a single `LISTEN` socket and fork 48 times, or fork 48 times, and each child process creates a `LISTEN` socket after enabling `SO_REUSEPORT`.

4. Client creates 48 processes. Each process connects and disconnects to the server a

```
 Performance counter stats for 'system wide':

    86,19,74,98,398      L1-icache-load-misses                                       (7.56%)
  1,10,52,87,96,998      L1-dcache-load-misses     #   13.36% of all L1-dcache hits  (10.18%)
  8,27,50,42,29,986      L1-dcache-loads                                             (8.10%)
  5,16,99,57,43,368      L1-dcache-stores                                            (8.55%)
 57,45,46,10,85,783      cycles:k                                                    (10.24%)
 32,95,11,87,95,913      instructions:k            #    0.57  insn per cycle         (13.52%)
       43,06,81,967      cache-misses:k            #    0.378 % of all cache refs    (11.31%)
  1,13,78,73,68,926      cache-references:k                                          (14.03%)
        9,25,98,669      LLC-store-misses                                            (16.32%)
    11,42,49,24,289      LLC-store                                                   (16.41%)
       32,93,27,396      LLC-load-misses           #    0.79% of all LL-cache hits   (6.56%)
    41,91,86,14,432      LLC-load                                                    (6.56%)
     5,49,33,22,650      branch-misses             #    0.95% of all branches        (9.84%)
  5,77,43,47,99,056      branches                                                    (13.13%)
                  0      alignment-faults
  5,01,58,49,34,270      dTLB-stores                                                 (11.10%)
       73,75,94,625      dTLB-store-misses                                           (10.51%)
  9,88,34,59,81,693      dTLB-loads                                                  (7.62%)
     5,82,03,80,253      dTLB-load-misses          #    0.59% of all dTLB cache hits  (8.30%)
           1,25,026      page-faults
       17,93,91,744      context-switches

      367.197205402 seconds time elapsed
```

Figure 2. Performance Counter Statistics without SO_REUSEPORT

```
# Event count (approx.): 7885138570074
#
# Overhead  Command      Shared Object        Symbol
# ........  ...........  ...........  ....................................
    39.67%  swapper      [kernel.kallsyms]    [k] intel_idle
     4.69%  my_server    [kernel.kallsyms]    [k] native_queued_spin_lock_slowpath
     1.08%  swapper      [kernel.kallsyms]    [k] menu_select
     0.90%  swapper      [kernel.kallsyms]    [k] cpuidle_enter_state
     0.81%  swapper      [kernel.kallsyms]    [k] __bpf_prog_run
     0.77%  swapper      [mlx4_en]            [k] mlx4_en_process_rx_cq
     0.76%  swapper      [kernel.kallsyms]    [k] fib_table_lookup
     0.69%  swapper      [kernel.kallsyms]    [k] native_queued_spin_lock_slowpath
     0.68%  my_server    [kernel.kallsyms]    [k] _raw_spin_lock_bh
     0.62%  swapper      [ip_tables]          [k] ipt_do_table
     0.61%  my_server    [kernel.kallsyms]    [k] update_load_avg
     0.60%  swapper      [mlx4_core]          [k] set_and_calc_slave_port_state
     0.58%  swapper      [kernel.kallsyms]    [k] fib_rules_lookup
     0.57%  swapper      [kernel.kallsyms]    [k] switch_mm_irqs_off
     0.54%  swapper      [kernel.kallsyms]    [k] packet_rcv
     0.49%  swapper      [kernel.kallsyms]    [k] __sched_text_start
     0.46%  my_server    [kernel.kallsyms]    [k] update_cfs_group
     0.44%  swapper      [kernel.kallsyms]    [k] do_idle
     0.43%  swapper      [kernel.kallsyms]    [k] native_irq_return_iret
     0.42%  my_server    [kernel.kallsyms]    [k] kmem_cache_free
     0.42%  my_server    [kernel.kallsyms]    [k] _raw_spin_lock_irqsave
     0.38%  my_server    [kernel.kallsyms]    [k] __sched_text_start
     0.38%  my_server    [kernel.kallsyms]    [k] kmem_cache_alloc
     0.37%  my_server    [kernel.kallsyms]    [k] inet_csk_accept
     0.37%  swapper      [kernel.kallsyms]    [k] _raw_spin_lock_irqsave
```

Figure 3. Performance Profile of the Top 25 Functions without SO_REUSEPORT

```
 Performance counter stats for 'system wide':

    76,04,13,62,686      L1-icache-load-misses                                       (10.89%)
    82,15,93,08,972      L1-dcache-load-misses     #   11.18% of all L1-dcache hits  (13.98%)
 7,34,96,16,22,399       L1-dcache-loads                                             (12.66%)
 3,82,80,75,66,845       L1-dcache-stores                                            (11.37%)
38,77,24,76,24,587       cycles:k                                                    (11.01%)
23,67,44,06,96,707       instructions:k            #    0.61  insn per cycle         (15.76%)
       23,50,65,625      cache-misses:k            #    0.286 % of all cache refs    (15.38%)
    82,11,60,50,254      cache-references:k                                          (19.34%)
        3,57,19,806      LLC-store-misses                                            (22.88%)
     7,02,98,51,892      LLC-store                                                   (23.78%)
       18,90,54,856      LLC-load-misses           #    0.63% of all LL-cache hits   (9.51%)
    29,98,94,35,477      LLC-load                                                    (9.51%)
     3,75,09,31,992      branch-misses             #    0.86% of all branches        (14.27%)
 4,35,66,38,46,056       branches                                                    (19.02%)
                 0       alignment-faults
 3,63,98,33,91,249       dTLB-stores                                                 (15.01%)
      44,78,16,924       dTLB-store-misses                                           (15.27%)
 7,17,35,00,47,536       dTLB-loads                                                  (10.33%)
    4,52,11,77,507       dTLB-load-misses          #    0.63% of all dTLB cache hits  (10.80%)
           51,308        page-faults
       7,24,86,433       context-switches

     150.580805600 seconds time elapsed
```

Figure 4. Performance Counter Statistics with SO_REUSEPORT

```
# Event count (approx.): 5449235671840
#
# Overhead  Command      Shared Object        Symbol
# ........  ...........  ...........  ....................................
    26.28%  swapper      [kernel.kallsyms]    [k] intel_idle
     0.92%  my_server    [kernel.kallsyms]    [k] kmem_cache_free
     0.87%  swapper      [mlx4_en]            [k] mlx4_en_process_rx_cq
     0.85%  my_server    [mlx4_en]            [k] mlx4_en_process_rx_cq
     0.82%  my_server    [kernel.kallsyms]    [k] __bpf_prog_run
     0.82%  my_server    [kernel.kallsyms]    [k] kmem_cache_alloc
     0.82%  swapper      [kernel.kallsyms]    [k] fib_table_lookup
     0.79%  my_server    [ip_tables]          [k] ipt_do_table
     0.78%  my_server    [kernel.kallsyms]    [k] fib_table_lookup
     0.76%  swapper      [kernel.kallsyms]    [k] __bpf_prog_run
     0.75%  my_server    [kernel.kallsyms]    [k] packet_rcv
     0.69%  my_server    [kernel.kallsyms]    [k] _raw_spin_lock
     0.64%  swapper      [kernel.kallsyms]    [k] fib_rules_lookup
     0.62%  swapper      [ip_tables]          [k] ipt_do_table
     0.61%  my_server    [kernel.kallsyms]    [k] __entry_trampoline_start
     0.60%  swapper      [kernel.kallsyms]    [k] packet_rcv
     0.59%  my_server    [kernel.kallsyms]    [k] syscall_return_via_sysret
     0.58%  my_server    [kernel.kallsyms]    [k] _raw_spin_lock_bh
     0.58%  swapper      [kernel.kallsyms]    [k] menu_select
     0.57%  my_server    [kernel.kallsyms]    [k] fib_rules_lookup
     0.53%  swapper      [kernel.kallsyms]    [k] cpuidle_enter_state
     0.48%  swapper      [mlx4_core]          [k] set_and_calc_slave_port_state
     0.45%  my_server    [mlx4_core]          [k] set_and_calc_slave_port_state
     0.43%  swapper      [kernel.kallsyms]    [k] _raw_spin_lock
     0.42%  swapper      [kernel.kallsyms]    [k] __netif_receive_skb_core
```

Figure 5. Performance Profile of the Top 25 Functions with SO_REUSEPORT

million times sequentially.

With the fork of the `LISTEN` socket:

```
server-system-$  ./my_server 45000  48  0   (0 indicates fork)
client-system-$  time ./my_client <server-ip> 45000 48 1000000
real   4m45.471s
```

With `SO_REUSEPORT`:

```
server-system-$ ./my_server 45000 48  1  (1 indicates
 ↪SO_REUSEPORT)
client-system-$ time ./my_client <server-ip> 45000 48 1000000
real   1m36.766s
```

Performance Analysis of SO_REUSEPORT

Figures 2–5 provide a look at the performance profile for the above two tests using the `perf` tool.

Source Code for Client and Server Applications

The listing below implements a server and client application that were used for `SO_REUSEPORT` performance testing:

```
$ cat my_server.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <netdb.h>

void create_children(int nprocs, int parent_pid)
{
   while (nprocs-- > 0) {
```

```
      if (getpid() == parent_pid && fork() < 0)
         exit(1);
   }
}

int main(int argc, char *argv[])
{
   int reuse_port, fd, cfd, nprocs, opt = 1, parent_pid =
    ↪getpid();
   struct sockaddr_in server;

   if (argc != 4) {
      fprintf(stderr, "Port# #Procs {0->fork, or
       ↪1->SO_REUSEPORT}\n");
      return 1;
   }

   nprocs = atoi(argv[2]);
   reuse_port = atoi(argv[3]);
   if (reuse_port)   /* proper SO_REUSEPORT */
      create_children(nprocs, parent_pid);

   if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      return 1;
   }

   if (reuse_port)
      setsockopt(fd, SOL_SOCKET, SO_REUSEPORT, (char *)&opt,
                 sizeof opt);

   server.sin_family      = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
```

```
   server.sin_port        = htons(atoi(argv[1]));

   if (bind(fd, (struct sockaddr *)&server, sizeof server)
    ↪< 0) {
      perror("bind");
      return 1;
   }

   if (!reuse_port)   /* simple fork instead of SO_REUSEPORT */
      create_children(nprocs, parent_pid);

   if (parent_pid == getpid()) {
      while (wait(NULL) != -1);   /* wait for all children */
   } else {
      listen(fd, SOMAXCONN);
      while (1) {
         if ((cfd = accept(fd, NULL, NULL)) < 0) {
            perror("accept");
            return 1;
         }
         close(cfd);
      }
   }

   return 0;
}

$ cat my_client.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <sys/wait.h>
```

```
#include <netdb.h>

void create_children(int nprocs, int parent_pid)
{
   while (nprocs-- > 0) {
      if (getpid() == parent_pid && fork() < 0)
         exit(1);
   }
}

int main(int argc, char *argv[])
{
   int fd, count, nprocs, parent_pid = getpid();
   struct sockaddr_in server;
   struct hostent *server_ent;
   const struct linger nolinger = { .l_onoff = 1,
    ↪.l_linger = 0 };

   if (argc != 5) {
      fprintf(stderr, "Server-IP Port# #Processes
       ↪#Conns_per_Proc\n");
      return 1;
   }

   nprocs = atoi(argv[3]);
   count = atoi(argv[4]);

   if ((server_ent = gethostbyname(argv[1])) == NULL) {
      perror("gethostbyname");
      return 1;
   }

   bzero((char *)&server, sizeof server);
```

```
   server.sin_family = AF_INET;
   server.sin_port   = htons(atoi(argv[2]));
   bcopy((char *)server_ent->h_addr, (char *)
↪&server.sin_addr.s_addr,
         server_ent->h_length);

   create_children(nprocs, parent_pid);

   if (getpid() == parent_pid) {
      while (wait(NULL) != -1);   /* wait for all children */
   } else {
      while (count-- > 0) {
         if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket");
            return 1;
         }

         if (connect(fd, (struct sockaddr *)&server,
                     sizeof server) < 0) {
            perror("connect");
            return 1;
         }

         /* Reset connection to avoid TIME-WAIT state */
         setsockopt(fd, SOL_SOCKET, SO_LINGER, &nolinger,
                    sizeof nolinger);
         close(fd);
      }
   }

   return 0;
}
```

Resources

- "The SO_REUSEPORT socket option" by Michael Kerrisk on LWN.net
- Network Socket (Wikipedia)
- Transmission Control Protocol (Wikipedia)
- TCP State Transition Diagram
- Kernel Source Code
- UNIX Network Programming by W. Richard Stevens, Bill Fenner and Andrew M. Rudoff
- Arrays of Length Zero
- Reciprocal Multiplication

Krishna Kumar works at Flipkart Internet Pvt Ltd, India's largest e-commerce company. He is especially interested in today's topic, as Flipkart uses this technology to host millions of connections from visitors all over the world. His other interests are playing chess, struggling to learn to use apps, and occasionally bringing stray puppies home much to his wife's consternation. Please send your comments and feedback to krishna.ku@flipkart.com.

Open Source—It's in the Genes

Glyn Moody has been writing about the internet since 1994, and about free software since 1995. In 1997, he wrote the first mainstream feature about GNU/Linux and free software, which appeared in *Wired*. In 2001, his book *Rebel Code: Linux And The Open Source Revolution* was published. Since then, he has written widely about free software and digital rights. He has a blog, and he is active on social media: @glynmoody on Twitter or identi.ca, and +glynmoody on Google+.

What happens when you release 500,000 human genomes as open source? This.

By Glyn Moody

DNA is digital. The three billion chemical bases that make up the human genome encode data not in binary, but in a quaternary system, using four compounds—adenine, cytosine, guanine, thymine—to represent four genetic "digits": A, C, G and T. Although this came as something of a surprise in 1953, when Watson and Crick proposed an A–T and C–G pairing as a "copying mechanism for genetic material" in their famous double helix paper, it's hard to see how hereditary information could have been transmitted efficiently from generation to generation in any other way. As anyone who has made photocopies of photocopies is aware, analog systems are bad at loss-free transmission, unlike digital encodings. Evolution of progressively more complex structures over millions of years would have been much harder, perhaps impossible, had our genetic material been stored in a purely analog form.

Although the digital nature of DNA was known more than half a century ago, it was only after many years of further work that quaternary data could be extracted at scale. The Human Genome Project, where laboratories around the world pieced together the three billion bases found in a single human genome, was completed in 2003, after 13 years of work,

for a cost of around $750 million. However, since then, the cost of sequencing genomes has fallen—in fact, it has plummeted even faster than Moore's Law for semiconductors. A complete human genome now can be sequenced for a few hundred dollars, with sub-$100 services expected soon.

As costs have fallen, new services have sprung up offering to sequence—at least partially—anyone's genome. Millions have sent samples of their saliva to companies like 23andMe in order to learn things about their "ancestry, health, wellness and more". It's exciting stuff, but there are big downsides to using these companies. You may be giving a company the right to use your DNA for other purposes. That is, you are losing control of the most personal code there is—the one that created you in the boot-up process we call gestation. Deleting sequenced DNA can be hard.

That's bad enough, but it gets worse. Because the DNA of all your relatives is similar to yours to varying degrees, when you have your genome sequenced, you are effectively giving away part of their DNA too. Whether they agree or not, they lose their genetic anonymity, which may have serious and unforeseen consequences. In the US, police are using genetic information that has been made public by individuals to find partial matches of DNA from a crime scene. By building and exploring family trees on a massive scale, the police can narrow their investigations down to a few suspects to help them pinpoint the criminal.

Just as software code can be open source rather than proprietary, so there are publicly funded genomic sequencing initiatives that make their results available to all. One of the largest projects, the UK Biobank (UKB), involves 500,000 participants. Any researcher, anywhere in the world, can download complete, anonymized data sets, provided they are approved by the UKB board. One important restriction is that they must not try to re-identify any participant—something that would be relatively easy to do given the extremely detailed clinical history that was gathered from volunteers along with blood and urine samples. Investigators asked all 500,000 participants about their habits, and examined them for more than 2,000 different traits, including data on their social lives, cognitive state, lifestyle and physical health.

Given the large number of genomes that need to be sequenced, the first open DNA data sets from UKB are only partial, although the plan is to sequence all genomes more fully in due course. These smaller data sets allow what is called "genotyping", which provides a rough map of a person's DNA and its specific properties. Even this partial sequencing provides valuable information, especially when it is available for large numbers of people. As an article in *Science* points out, it is not just the size and richness of the open data sets that makes the UK Biobank unique, it is the thorough-going nature of the sharing that is required from researchers:

> Researchers around the world can freely delve into the UKB data and rapidly build on one another's work, resulting in unexpected dividends in diverse fields, such as human evolution. In a crowdsourcing spirit rare in the hypercompetitive world of biomedical research, groups even post tools for using the data without first seeking credit by publishing in a journal.

The benefits from applying open-source methodology to half a million genomes are significant and growing by the day. About 7,000 researchers have registered to use UKB data on 1,400 projects, and more than 600 papers have been published. It is leading to rapid advances that are simply not possible when the DNA is proprietary. And as with open source, doing good brings benefits:

> "The U.K. is getting all of the world's best brains" to study its citizens, says Ewan Birney, director of the EMBL European Bioinformatics Institute in Hinxton, U.K., and a member of the UKB's steering committee. The U.K. focus is also the project's chief downside, as it explores just one slice of humanity: northern Europeans. It holds data for only about 20,000 people of African or Asian descent, for example. Yet as new papers appear every few days, researchers say the UKB remains a shining example of the power of curiosity unleashed. "It's the thing we always dreamed of," [president and director of the Broad Institute in Cambridge, Massachusetts] Lander says.

It's the classic "given enough eyeballs, all bugs are shallow". By open-sourcing the genomic code of 500,000 of its citizens, the UK is getting the top DNA hackers in the

world to find the “bugs”—the variants that are associated with medical conditions—that will help our understanding of them and may well lead to the development of new treatments for them. The advantages are so obvious, it’s a wonder people use anything else. A bit like open source. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.