Review: elementary 5 "Juno"

Thoughts on Open Core

Best of Linux Marketing Campaigns

LINUX JOURNAL

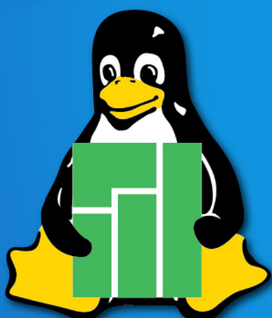
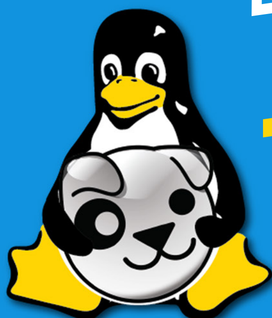
Since 1994: The original magazine of the Linux community

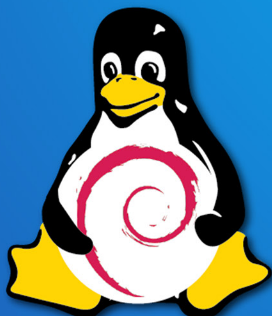
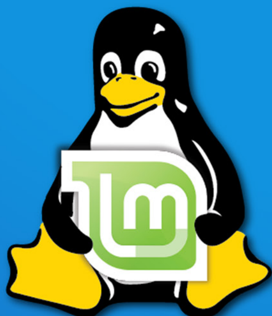
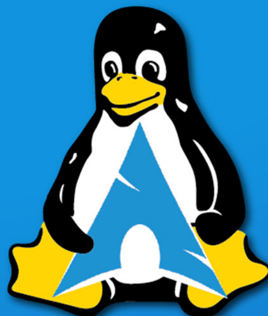
DISTRIBUTIONS

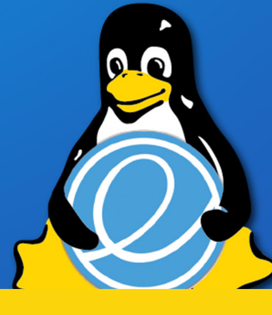
LINUX DISTRO ROUNDUP

+ THE RICH DIVERSITY OF DISTROS

BUILD YOUR OWN CUSTOM DISTRO

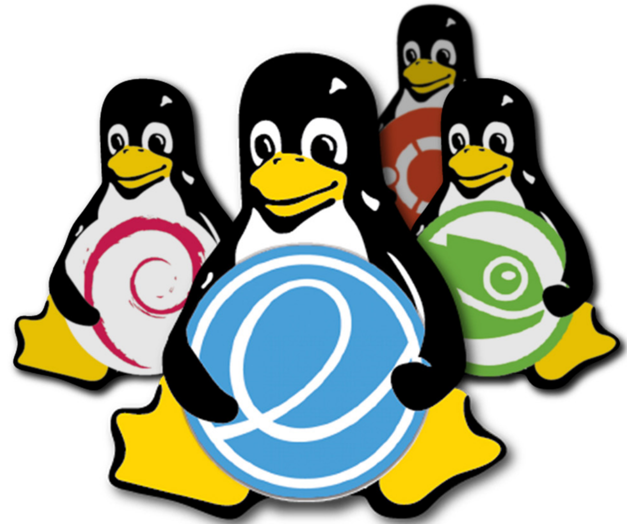


ISSUE 294 | JANUARY 2019
https://www.linuxjournal.com

CONTENTS

JANUARY 2019
ISSUE 294

72 DEEP DIVE: *Distributions*



73 The State of Desktop Linux 2019
by Bryan Lunduke
A snapshot of the current state of Desktop Linux at the start of 2019 with comparison charts and a roundtable Q&A with the leaders of three top Linux distributions.

92 Linux and the Multiverse
by Marcel Gagné
A look at the rich diversity of Linux distributions.

104 Build a Custom Minimal Linux Distribution from Source, Part II
by Petros Koutoupis
Follow along with this step-by-step guide to creating your own distribution.

121 elementary 5 "Juno"
by Bryan Lunduke
A review of the elementary distribution and an interview with its founders.

6 **The Distribution Issue**
by Bryan Lunduke

10 **From the Editor—Doc Searls**
Where There's No Distance or Gravity

UPFRONT

16 **Best Linux Marketing Campaigns**
by Bryan Lunduke

20 **Modeling the Entire Universe**
by Joey Bernard

26 **Patreon and Linux Journal**

27 **Some Thoughts on Open Core**
by Kyle Rankin

30 **Put Down the Pipe**
by Kyle Rankin

33 **FOSS Project Spotlight: Mender.io, an Open-Source Over-the-Air Software Update Manager for IoT Devices**
by Ralph Nguyen

37 **Reality 2.0: a Linux Journal Podcast**

38 **News Briefs**

COLUMNS

42 **Kyle Rankin's Hack and /**
Back to Basics: Sort and Uniq

48 **Reuven M. Lerner's At the Forge**
Python Testing with pytest: Fixtures and Coverage

54 **Dave Taylor's Work the Shell**
Converting Decimals to Roman Numerals with Bash

63 **Zack Brown's diff -u**
What's New in Kernel Development

157 **Glyn Moody's Open Sauce**
IBM Began Buying Red Hat 20 Years Ago

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.

ARTICLE

138 A Use Case for Network Automation by Eric Pearce
Use the Python Netmiko module to automate switches, routers and firewalls from multiple vendors.

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 +1-281-944-5188.

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

GRAPHIC DESIGNER: Garrick Antikajian, garrick@linuxjournal.com

ACCOUNTANT: Candy Beauchamp, acct@linuxjournal.com

COMMUNITY ADVISORY BOARD
John Abreau, Boston Linux & UNIX Group; John Alexander, Shropshire Linux User Group;
Robert Belnap, Classic Hackers UGA Users Group; Aaron Chantrill, Bellingham Linux Users Group;
Lawrence D'Oliveiro, Waikato Linux Users Group; Chris Ebenezer, Silicon Corridor Linux User Group;
David Egts, Akron Linux Users Group; Michael Fox, Peterborough Linux User Group;
Braddock Gaskill, San Gabriel Valley Linux Users' Group; Roy Lindauer, Reno Linux Users Group;
Scott Murphy, Ottawa Canada Linux Users Group; Andrew Pam, Linux Users of Victoria;
Bob Proulx, Northern Colorado Linux User's Group; Ian Sacklow, Capital District Linux Users Group;
Ron Singh, Kitchener-Waterloo Linux User Group; Jeff Smith, Kitchener-Waterloo Linux User Group;
Matt Smith, North Bay Linux Users' Group; James Snyder, Kent Linux User Group;
Paul Tansom, Portsmouth and South East Hampshire Linux User Group;
Gary Turner, Dayton Linux Users Group; Sam Williams, Rock River Linux Users Group;
Stephen Worley, Linux Users' Group at North Carolina State University;
Lukas Yoder, Linux Users Group at Georgia Tech

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: Publisher Carlie Fairchild
Phone: +1-281-944-5188

LINUX is a registered trademark of Linus Torvalds.

privateinternetaccess®
always use protection®
Private Internet Access is a proud sponsor of *Linux Journal*.

Join a community with a deep appreciation for open-source philosophies, digital freedoms and privacy.

Subscribe to *Linux Journal* Digital Edition *for only* ***$2.88 an issue.***

SUBSCRIBE TODAY!

THE DISTRIBUTIONS ISSUE

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*.

Do you remember your first distro?

By Bryan Lunduke

The first version of Linux I truly used, for any length of time, was back at the end of the 1990s—in *Ye Olden Times*, when 56k modems, 3.5” floppies and VGA CRT monitors reigned supreme.

Linux itself had been a thing for a number of years by that point—with both SUSE (then known as the gloriously mixed-case and punctuation-filled S.u.S.E.) and Red Hat doing good business supporting it—when I decided to really give this “Free” operating system a try.

Because I'm a nerd. And that's what we do.

I remember the day well. It was cold. It was rainy. And I was taking an extended lunch break from my job at Microsoft (seriously). My days—and, all too often, nights—spent testing Windows NT 5 (before it was renamed Windows 2000) had taken a toll. I had reached peak “burn out”.

After a mildly rejuvenating, two-hour long, burger-eating (and venting about our job) session with a co-worker, we made our way to the big-box computer store close to Microsoft's main campus. Once inside, we bee-lined it for the Operating System section (this was back when

computer stores had rows upon rows of actual boxes that contained actual physical media, which, in turn, contained actual software).

Several versions of Windows were on display, and, lo and behold, right there next to them, was S.u.S.E. Linux—in a box. I grabbed it immediately. It was heavy. There were several CDs inside along with a manual (which would turn out to be necessary simply to get the system to boot).

Fifteen minutes later, we were back in my office installing Linux on one of my little Dell towers.

That's right. My first full-time Linux machine? A Microsoft, company-issued work computer. This was my way of "sticking it to the man"—and boy did it feel good.

Were there problems with my first foray into Linux? You bet. The sound card didn't work. Getting an X Server running (with any sort of GUI) was a mildly mystifying process. And, heck, just getting the darn thing to boot took the better part of an afternoon. But, even with those challenges, I was in love.

Thus, my 20-year long hobby of "installing every Linux distribution I can get my grubby little hands on" was born—right there on Microsoft's main campus, using funds I earned from my job at Microsoft, on Microsoft-owned hardware, using Microsoft-supplied electricity and company time.

Shh. Don't tell Ballmer.

From that point onward, one of the things about Linux that always has made me smile is the wide variety of distributions out there in the world. There seems to be one custom-made for every man, woman and child on planet Earth.

In this issue of *Linux Journal*, Marcel Gagné takes a look at some of the more interesting aspects of this in a lovely piece titled "Linux and the Multiverse", comparing the diverse world of Linux distros with the origins of our own, physical universe. And the Multiverse—somehow, Hannah Montana Linux is part of that. It'll make sense when you read it.

Also in this issue, I describe sitting down with the project leaders of three prominent Linux distributions—Debian, Fedora and elementary—to have a bit of a round-table, Q&A-styled discussion about the broader Linux distribution ecosystem, marketshare and the challenges we face. It's truly fascinating watching these three leaders of our industry present their—sometimes differing, sometimes similar—views.

Since I was talking to project leaders, I pull aside the elementary founders to talk about their latest release (version 5.0, code-named "Juno") and what they've got in store for the future.

But...what if you want to build your own Linux distribution? Like *really* build it. From scratch.

Our own Petros Koutoupis provides a detailed, step-by-step guide to doing exactly that in his perfectly named "Build a Custom Minimal Linux Distribution from Source, Part II". He walks you through getting everything built, from source code, to have a fully functioning system and web server.

It's an incredibly satisfying process. Highly recommend.

Now, if you'll excuse me, there's got to be a distro out there I haven't installed yet. ■

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

Thanks to Sponsor
PULSEWAY
for Supporting *Linux Journal*

System Management
at Your Fingertips.

www.pulseway.com

Want to see your company's logo here?
Find out more, **https://www.linuxjournal.com/sponsors**.

Where There's No Distance or Gravity

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.

The more digital we become, the less human we remain.

By Doc Searls

I had been in Los Angeles only a few times in my life before the October day in 1987 when I drove down from our home in the Bay Area with my teenage son to visit family. The air was unusually clear as we started our drive back north, and soon the San Gabriel Mountains—Los Angeles' own Alps (you can ski there!)—loomed over the region like a crenelated battlement, as if protecting its inhabitants from cultures and climates that might invade from the north. So, on impulse, I decided to drive up to Mount Wilson, the only crest in the range with a paved road to the top.

I could see from the maps I had already studied that the drive was an easy one. Our destination also was easily spotted from below: a long, almost flat ridge topped by the white domes of Mount Wilson Observatory (where Hubble observed the universe expanding) and a bristle of towers radiating nearly all the area's FM and TV signals. The site was legendary among broadcast engineering geeks, and I had longed to visit it ever since I was a ham

radio operator as a boy in New Jersey.

After checking out the observatory and the towers, my son and I stood on a promontory next to a parking lot and surveyed the vast spread of civilization below. Soon four visiting golfers from New York came over and started asking me questions about what was where.

I answered like a veteran docent, pointing out the Rose Bowl, Palos Verdes Peninsula, Santa Catalina and other Channel Islands, the Hollywood Hills, the San Fernando Valley, the Jet Propulsion Laboratory, Santa Anita Park and more. When they asked where the Whittier Narrows earthquake had happened a few days before, I pointed at the Puente Hills, off to the southeast, and filled them in on what I knew about the geology there as well.

After a few minutes of this, they asked me how long I had lived there. I said all this stuff was almost as new to me as it was to them. “Then how do you know so much about it?”, they asked. I told them I had studied maps of the area and refreshed my knowledge over lunch just before driving up there. They were flabbergasted. “Really?”, one guy said. “You study maps?”

Indeed, I did. I had maps of all kinds and sizes at home, and the door pockets of my

car bulged with AAA maps of everywhere I might drive in California. I also added local and regional Southern California maps to my mobile inventory before driving down.

My obsession with maps, and my dependence on them, comported with what a shrink at a party once told me about obsessive compulsive disorder, aka OCD: that it accounted both for most mental illnesses and for nearly all of humanity's great achievements.

While the achievements in my case might not have been great, obsessively reading maps got me into geography, geology, astronomy and much of the rest of what I know about nature and technology.

What got me interested in maps was radio. My interest there was less in radio's entertainment value than in how signals worked. That began with wondering what was happening under the blinking red lights on towers that stood in the swampland between our house in New Jersey and the New York skyline. Nearly all New York's AM stations broadcast from towers in those swaps, mostly to take advantage of both cheap land and salt water (which AM signals love). My idea of a good time was to ride my bike down there and visit with engineers staffing the transmitters (which these days require scant human attention).

I examined signals mostly with my Hammarlund HQ-129X ham radio receiver, connected to a 40-meter dipole antenna. (I also had an 80 and a 15, and all were strung to trees in our back yard like a giant spider web.)

In addition to what I learned from beeping in Morse code to hams as far away as Sweden, I logged more than 800 AM stations (roughly eight per channel) and jammed a sewing pin for each one into a big map on a bulletin board.

With the help of broadcast engineering manuals (full of maps), I learned about ground conductivity (affecting AM range along the ground), skywave propagation (affecting distance reception on AM and shortwave), directional signals (aimed by multiple towers on AM and fancy antenna designs on shortwave), and most important by the inverse square law. That law explained why the strength of a radio signal (also of sound and light) was inversely proportional to the square of the distance

from the source, which meant signal strength declined across distance roughly on an asymptotic curve.

Later, as I dug into TV and FM signals (which use the VHF and UHF broadcast bands), I added fun learnings about other stuff broadcast engineering can teach, such as the dielectric (or capacitive) properties of atmosphere and how those contribute to tropospheric bending of signals. Thanks to "tropo" over the Pacific Ocean, on most days, I can watch TV and listen to FM signals from San Diego and Tijuana—here in Santa Barbara, 220+ miles away.

But mostly I don't. To explain why, I submit this conversation I had with my younger son, who was a teenager when we walked back from a (soon to be doomed) Radio Shack store in New York. We went there to buy a kitchen radio for an apartment we were renting, but the store had no radios. Although not verbatim, this was how it went:

"Radio is dead", he said.

"Why?", I replied.

"Because it's obsolete."

"Why?"

"I mean, look: what is the point of 'range' and 'coverage'?"

"Huh?"

"I mean, what's the point of a station fading away when you leave town?"

"Those are features, not bugs. Geography limits range. So do transmitter powers and the inverse square law. Also, you don't want stations interfering with each other."

"But all the stations in the world are all on the internet. You can get them on your phone and none of them interfere with each other there. On top of that, there's music streaming and podcasts."

I'm with him on that now. Nearly all my consumption of what we now call "content" is through glowing rectangles connected to the internet. So he and I are alike that way, but we're not alike in our knowledge of the physical world. Mine is informed by experience with maps and analog tech. His is informed mostly by what he gets through his own glowing rectangles.

Marshall McLuhan said all technologies are extensions of our bodies and minds. He also said they shape us after we shape them, and that all new technologies "work us over completely".

Glowing rectangles have replaced paper maps in my life, along with radios and TVs. While losing those has changed how I understand and navigate the physical world, what I've gained, along with everybody else connected by the internet, is residence in a habitat absent of gravity and distance. Sure, there there are propagation delays on the net (such as those shown by `ping` and `traceroute` commands), and connections can get sphinctered in places (or filtered fully, as they are in China). But our experience of being present in the networked world is one of absent distance and of placelessness (and hence of gravity).

We can't help attempting to reify this virtual world with metaphors borrowed from the physical one: *sites*, *domains* and *locations*, for example. But they're misleading. We really don't "surf" or "visit" or "browse" those places (which aren't). We request files, which get copied onto our screens, without any sense of a distance having been traveled. Yet we collectively imagine that these are real sites, and real property, and that we are somehow mere visitors allowed by their owners to trespass on them. And, because of that assumption, many liberties are taken with our private digital selves by the operators of those sites and by the third parties they also bring in. These are privacy abuses we never would welcome or allow in the physical world. But they are normative in the virtual one.

For now.

The networked world we have today has been with us only a little more than two decades. That's very little time in which to start operating in fully civilized ways.

My wife, who (far as I know) was the first to observe that the internet gives us a second world without gravity or distance, insists that we can adapt, much as astronauts learn to live and work in a zero-gravity habitat. She also believes we are very early in the process of understanding this world, even as all both occupy it and continue to build it.

My point with the Mount Wilson story is that none of us are even close to having equivalents of the tools we use for understanding the physical world. Even the map metaphor is misleading. Where and how we live in the networked world is too different, too other, too singular. Like the universe, there are no other examples of it. There is just one of it.

Even if the networked world seems to start breaking up (as we're already seeing with China), at a deeper level that world arises from a simple protocol, TCP/IP, that isn't going away soon.

And even if TCP/IP gets replaced, the genie it liberated from the digital bottle won't stop giving everyone with a decent connection the experience of being together in a world without distance or gravity.

We also won't stop wanting to live in what John Updike (in the 1960s!) called "the age of full convenience". We can get full convenience only from networks that are completely free (as in freedom) and open to whatever.

We never will fully understand nor explain that world, any more than we ever will fully understood or explain the physical one. But we have much farther to go than we've come so far here on *terra firma*, especially since we're still terraforming it. And one of our jobs here is doing for this new world what maps did for the old one. ■

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

Best Linux Marketing Campaigns

I have long held the opinion that one of the biggest problems holding back Linux-based systems from dominating (market-share-wise) in the desktop computing space...is marketing. Our lack of attention-grabbing, hearts-and-minds-winning marketing is, in my oh-so-humble opinion, one of the most glaring weaknesses of the Free and Open Source Software world.

But, in a way, me saying that really isn't fair.

The reality is that we have had some truly fantastic marketing campaigns through the years. A few even managed to break outside our own Linux-loving community. Let's take a stroll through a few of my favorites.

From my vantage point, the best marketing has come from two places: IBM (which is purchasing Red Hat) and SUSE. Let's do this chronologically.

IBM's "Peace. Love. Linux."

Back in 2001, IBM made a major investment in Linux. To promote that investment, obviously, an ad campaign must be launched! Something iconic! Something catchy! Something...potentially illegal!

Boy, did they nail it.

"Peace. Love. Linux." Represented by simple symbols: a peace sign, a heart and a penguin, all in little circles next to each other. It was visually pleasing, and it promoted happiness (or, at least, peace and love). Brilliant!

IBM then paid to have more than 300 of these images spray-painted across sidewalks all over San Francisco. The paint was supposed to be biodegradable and wash away quickly. Unfortunately, that didn't happen—many of the stencils still were there months later.

And, according to the mayor, "Some were etched into the concrete, so, in those cases, they will never be removed."

The response from the city was...just as you'd expect.

After months of discussion, the City of San Francisco fined Big Blue $100,000, plus any additional cleanup costs, plus legal fees.

On the flip-side, the stories around it made for a heck of a lot of advertising!

IBM's "The Kid"

Remember the Linux Super Bowl ad from IBM? The one with the little boy sitting in a room of pure white light?

"He's learning. Absorbing. Getting smarter every day."

When that hit in 2004, it was like, *whoa*. Linux has made it. IBM made a Super Bowl ad about it!

"Does he have a name? His name...is Linux."

That campaign included Penny Marshall and Muhammad Ali. That's right. Laverne from *Laverne & Shirley* endorsed Linux in a Super Bowl ad. Let that sink in for a moment.

This was mind-blowing in 2004. Heck. It's kind of mind-blowing in 2019.

Novell's "PC, Mac & Linux"

Remember those "I'm a Mac" commercials from Apple? One guy ("Mac") poking

fun at how boring another guy ("PC") is? Well, Novell—which, you might recall, had purchased Linux company SuSE (back when the "U" was lowercase) a few years earlier—added a nice lady named "Linux" to the mix in 2007.

And, the results were kind of adorable. The videos had a decidedly "homemade but really well" feel to them. Every Linux podcast, blog and magazine talked about those little videos for a solid month after they were released.

SUSE's Music Videos

In the past few years, SUSE started regularly making parody music videos, and some of them are absolutely fantastic.

(*Full disclosure: I used to work for SUSE—specifically the marketing department of SUSE. More specific still, I wrote the lyrics to some of these songs. There's a slim possibility that I am mildly biased.*)

SUSE's music video adventure really kicked off in 2013 with a parody of Ylvis' "What Does the Fox Say?"—the aptly titled "What Does the Chameleon Say?". It was simple, dorky, fun and charming.

(*Note: the guy in the chameleon costume? He doesn't actually work for SUSE, but his dad is the video producer behind all of these videos, and he got roped in. You'll note that he plays the chameleon in a pretty large number of the music videos. Huzzah for consistency!*)

The most popular of the SUSE music videos came as a parody of Mark Ronson and Bruno Mars' "Uptown Funk"—"Uptime Funk." Believed to be the first music video about live-patching a Linux kernel on a running server, that 2015 song brought in a lot of attention both within and outside of the Linux community.

Beyond being the most popular, that one is also my personal favorite. And, yeah. I wrote it. I'm biased.

There are, of course, more. More music videos. More fun print and video ads. But these are the ones that have stood out to me through the years. The ones that felt *noteworthy*, like a landmark has been reached in the continual quest of spreading the word about Linux to the masses.

—Bryan Lunduke

Modeling the Entire Universe

For this article, I want to look at the largest thing possible, the whole universe. At least, that's the claim made by Celestia, the software package I'm introducing here. In all seriousness though, Celestia is a very well done astronomical simulator, similar to other software packages like Stellarium. Celestia is completely open source and is licensed under the GPL.

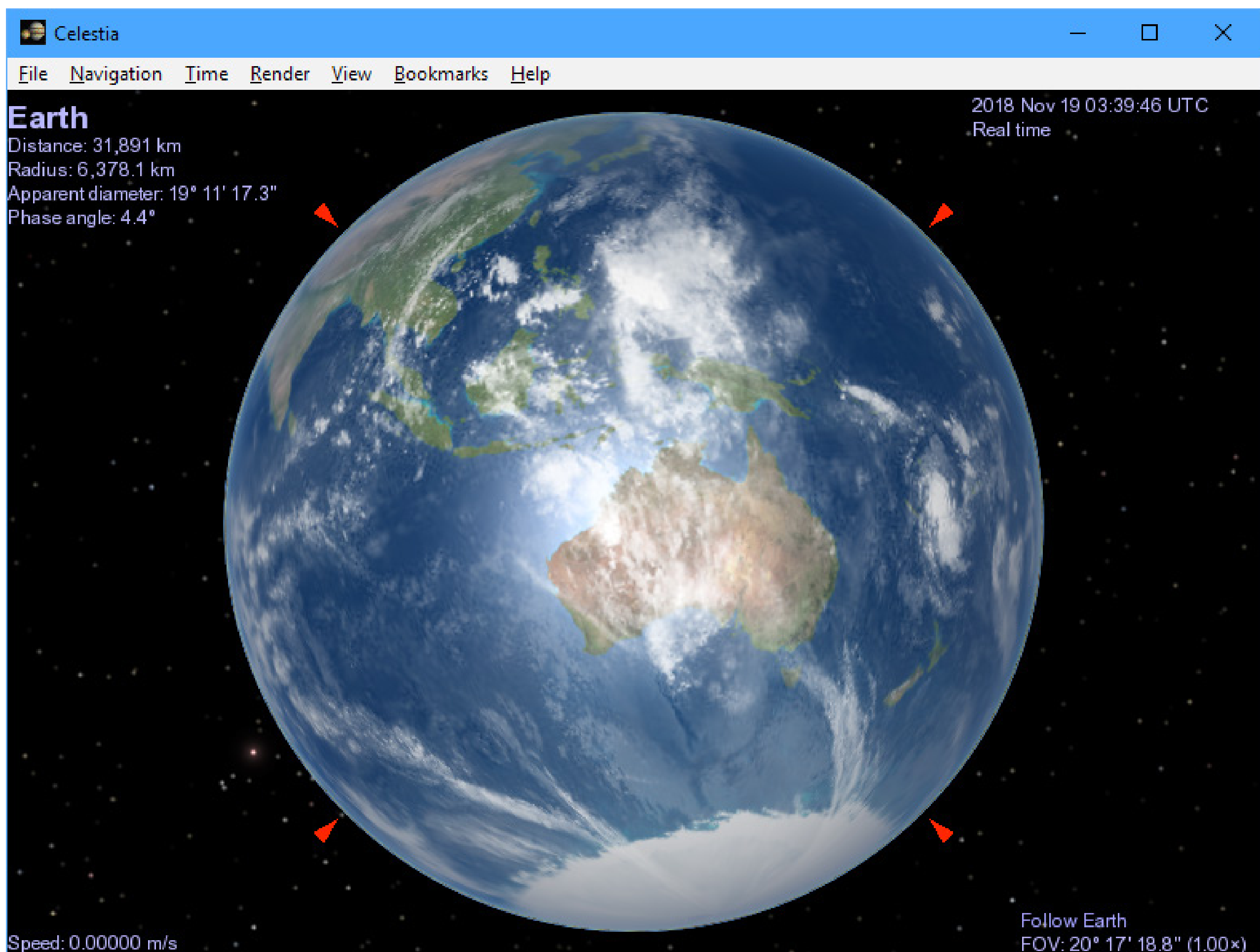


Figure 1. Celestia begins your exploration of space with a 3D view of Earth.

If Celestia isn't available via the package management system for your favorite distribution, you always can get the latest stable version from the Celestia's **website** as an installable binary package. If you really need the absolute latest version, you can grab it from the GitHub repository. Binaries also are available for Windows and Mac OS X, in case you need to travel on the dark side of computing.

Once you have installed Celestia, starting it provides a view of the Earth from space.

You're first placed on a track that follows the Earth through space. This is necessary, because Celestia is actually a real-time simulation. If you were in a fixed location in space, any object you were looking at quickly would leave your field of view. You can pause the simulation by pressing the spacebar. Once you are following an object,

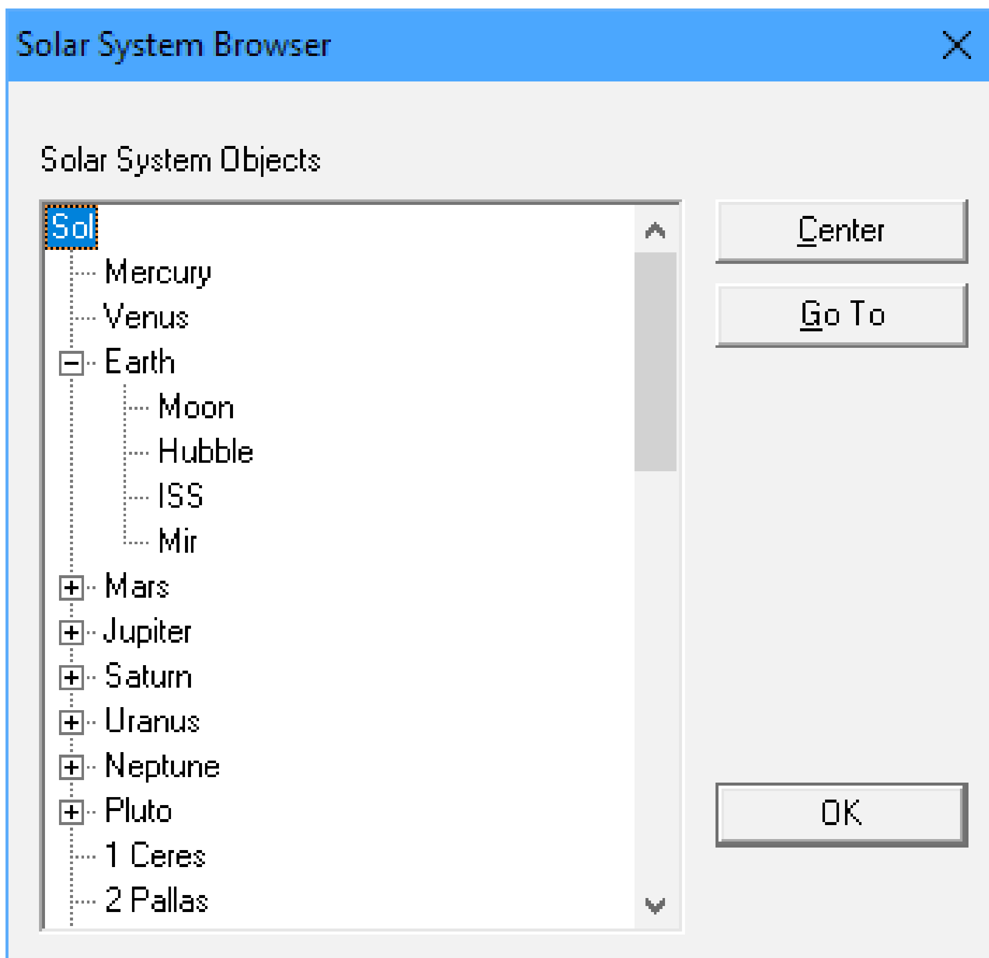


Figure 2. You can use the solar system browser to select objects to center on within the solar system.

you can rotate your view by clicking the left mouse button and dragging left/right or up/down.

If you're more interested in observing a centered object, you can click the right mouse button, and then dragging will move you around the object instead, allowing you to see the object's details. You can zoom in or out by using the mouse wheel. All of these navigation actions also have keyboard shortcuts, for those who prefer that to using a mouse.

But, how do you select which object you are centered on? The easiest option is to click the Navigation→Solar System Browser menu item to pop up a selection window.

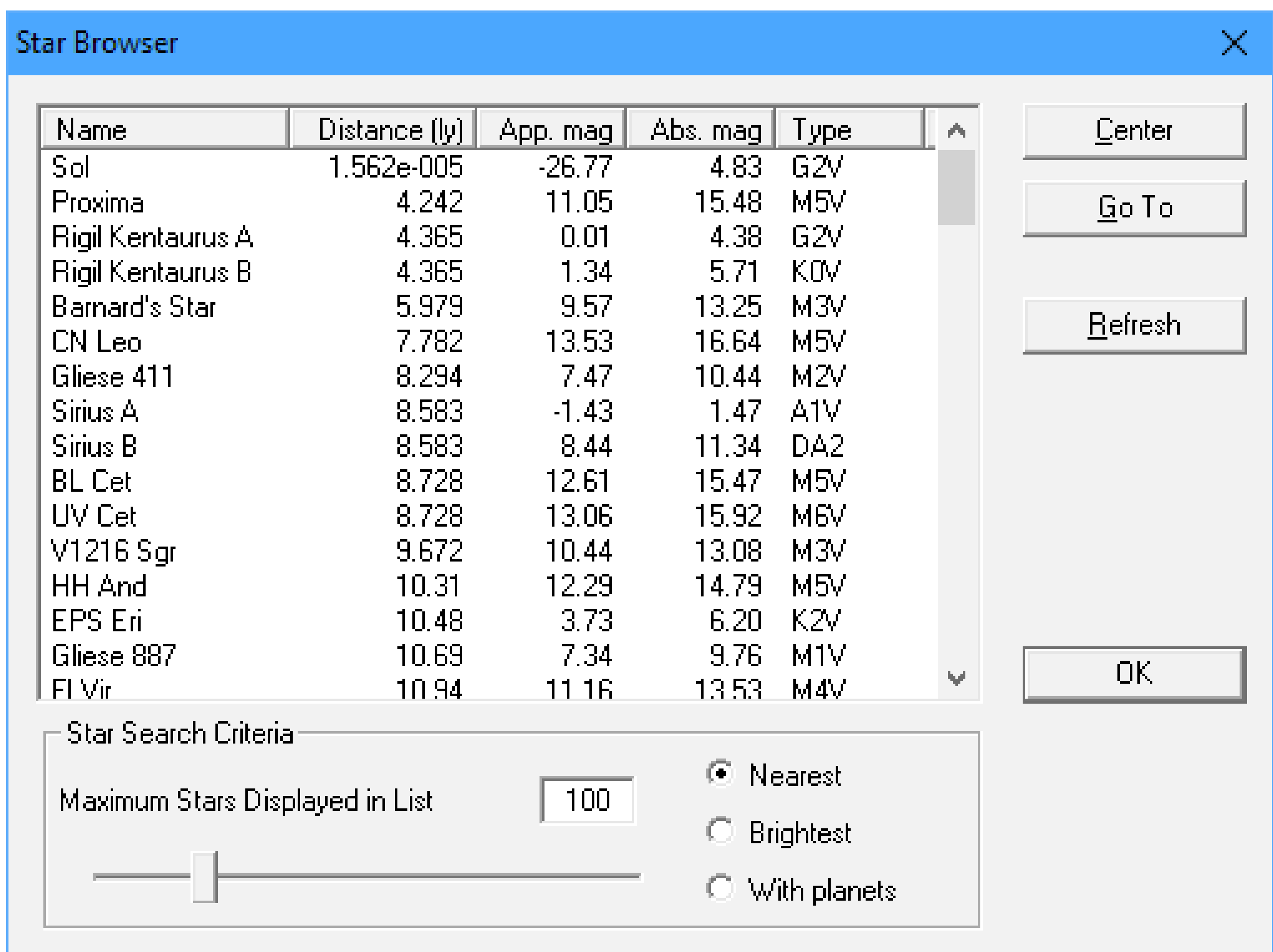


Figure 3. You can view objects beyond the solar system by bringing up the star browser window.

From here, you can choose from planets, moons, asteroids and other solar system objects available by default within Celestia (I'll explain how to add even more items shortly).

If you're looking at items beyond the solar system, you can click the Navigation→Star Browser menu item to open a new window.

From here, you can select from a large number of stars that are available in the standard library. If you want to go to a specific object or a specific location, click the Navigation→Goto Object menu item to open an input dialog where you can enter the details of where you want to go.

Until now, all of the objects that are available for viewing come with the standard installation of Celestia. However, Celestia also includes the ability to add extra items to the catalog. You can add object files for these additional objects in the extras subdirectory where Celestia is installed.

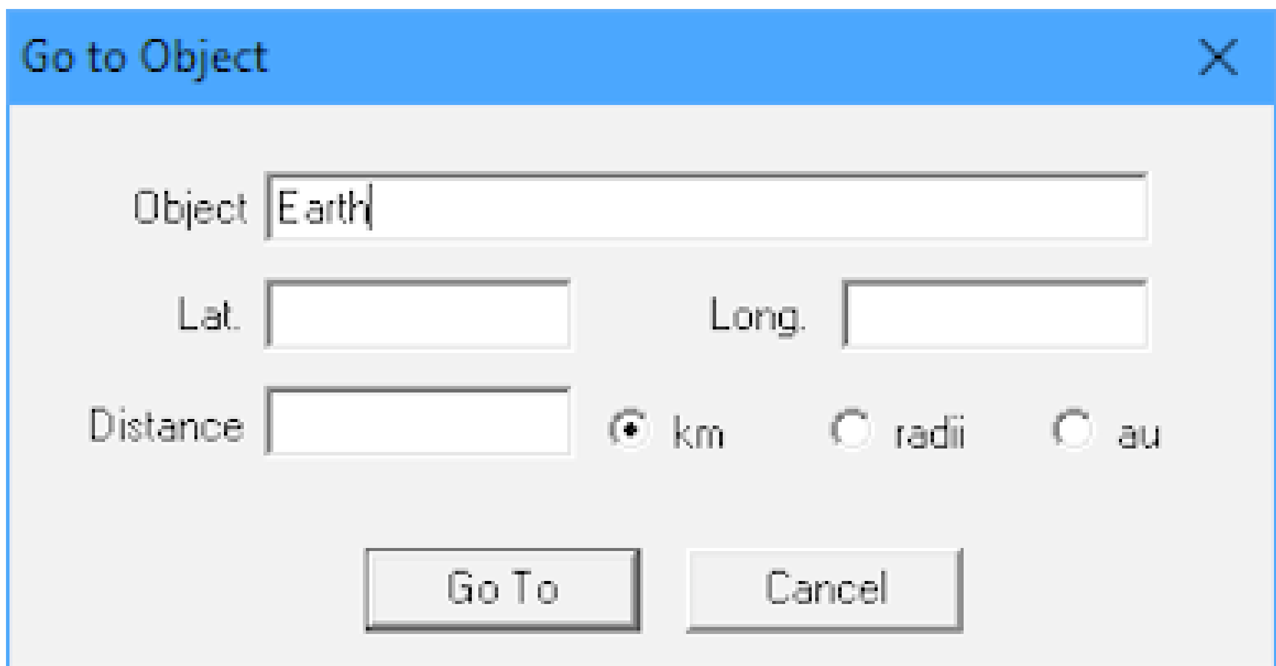


Figure 4. You can go to specific locations within the universe.

Several available objects are hosted at the Celestia Motherlode website. These are zip files, containing everything you need if you want to include that object in your installation of Celestia. You also can create your own extra objects and upload them to the Celestia Motherlode site in order to share them with other users.

You mostly interact with Celestia via text files. You can define how it behaves at start up by editing the start.cel and celestia.cfg files. These files are well commented, so you should be able to tune the way Celestia behaves relatively easily.

This interaction extends to being able to script Celestia, which is handy if you want to use it to create guided tours of celestial objects to show other people. These scripts are text files, with the filename ending in .celx. There's a complete scripting language that allows you to control most aspects of Celestia.

Once you have a certain view prepared, there are a few ways to share it with others. If you click File→Capture Image, a pop-up window appears where you can save the

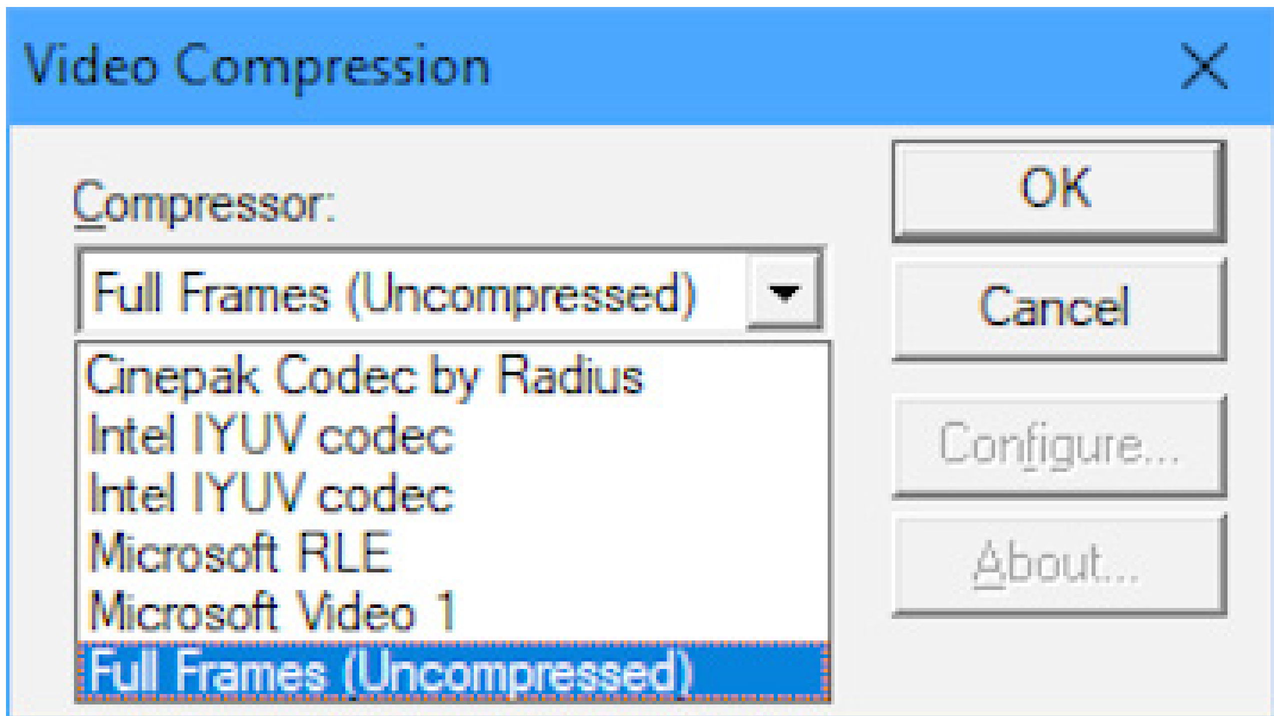


Figure 5. You can choose the video compression scheme to use when you record a video from Celestia.

currently rendered view as either a JPEG or PNG image file. Clicking File→Capture Movie opens a window where you can record a video file of what's occurring on the screen right then. You also can select the compression scheme to use or leave it as raw video.

This is handy if you want to share a tour of the universe with someone who may not have Celestia installed. However, instead of sharing images or videos, you also can share something called a Celestia URL (or Cel: URL). This URL includes the details of the rendered view, but when you do share it, the other person needs to have Celestia installed. One thing to be aware of is that there are some incompatibilities between versions of Celestia, so you may need to coordinate with the other person if you run into any issues.

Celestia should make a great addition to your astronomy toolkit. Its ability to script views is especially useful if you want to share something with students. See also the Celestia Wikibook for more information.

—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
- Black Baron
- Chris Short
- Christel Dahlskjaer
- David Breakey
- Dr. Stuart Makowski
- James Mayes
- James Weatherell
- Josh Simmons
- Magnus Magicman
- Mostly_Linux
- NDCHost.com
- Robert J. Hansen

Some Thoughts on Open Core

Why open core software is bad for the FOSS movement.

Nothing is inherently anti-business about Free and Open Source Software (FOSS). In fact, a number of different business models are built on top of FOSS. The best models are those that continue to further FOSS by internal code contributions and that advance the principles of Free Software in general. For instance, there's the support model, where a company develops free software but sells expert support for it.

Here, I'd like to talk a bit about one of the more problematic models out there, the open core model, because it's much more prevalent, and it creates some perverse incentives that run counter to Free Software principles.

If you haven't heard about it, the open core business model is one where a company develops free software (often a network service intended to be run on a server) and builds a base set of users and contributors of that free code base. Once there is a critical mass of features, the company then starts developing an "enterprise" version of the product that contains additional features aimed at corporate use. These enterprise features might include things like extra scalability, login features like LDAP/Active Directory support or Single Sign-On (SSO) or third-party integrations, or it might just be an overall improved version of the product with more code optimizations and speed.

Because such a company wants to charge customers to use the enterprise version, it creates a closed fork of the free software code base, or it might provide the additional proprietary features as modules so it has fewer problems with violating its free software license.

The first problem with the open core model is that on its face it doesn't further principles behind Free Software, because core developer time gets focused instead of writing and promoting proprietary software. Instead of promoting the importance of the freedoms that Free Software gives both users and developers, these companies often just use FOSS as a kind of freeware to get an initial base of users and as free crowdsourcing for software developers that develop the base product when the company is small and cash-strapped. As the company get more funding, it's then able to hire the most active community developers, so they then can stop working on the community edition and instead work full-time on the company's proprietary software.

This brings me to the second problem. The very nature of open core creates a perverse situation where a company is incentivized to put developer effort into improving the proprietary product (that brings in money) and is de-incentivized to move any of those improvements into the Free Software community edition. After all, if the community edition gets more features, why would someone pay for the enterprise edition? As a result, the community edition is often many steps behind the enterprise edition, if it gets many updates at all.

All of those productive core developers are instead working on improving the closed code. The remaining community ends up making improvements, often as (strangely enough) third-party modules, because it can be hard to get the company behind an open core project to accept modules that compete with its enterprise features.

What's worse is that a lot of the so-called "enterprise" features end up being focused on speed optimizations or basic security features like TLS support—simple improvements you'd want in the free software version. These speed or security improvements never make their way into the community edition, because the company intends that only individuals will use that version.

The message from the company is clear: although the company may support free software on its face (at the beginning), it believes that free software is for hobbyists and proprietary software is for professionals.

The final problem with the open core model is that after these startups move to the enterprise phase and start making money, there is zero incentive to start any *new* free software projects within the company. After all, if a core developer comes up with a great idea for an improvement or a new side project, that could be something the company could sell, so it winds up under the proprietary software "enterprise" umbrella.

Ultimately, the open core model is a version of Embrace, Extend and Extinguish made famous by Microsoft, only designed for VC-backed startups. The model allows startups to embrace FOSS when they are cash- and developer-strapped to get some free development and users for their software. The moment they have a base product that can justify the next round of VC funding, they move from embracing to extending the free "core" to add proprietary enterprise software. Finally, the free software core gets slowly extinguished. Improvements and new features in the core product slow to a trickle, as the proprietary enterprise product gets the majority of developer time and the differences between the two versions become too difficult to reconcile. The free software version becomes a kind of freeware demo for enterprise users to try out before they get the "real" version. Finally, the community edition lags too far behind and is abandoned by the company as it tries to hit the profitability phase of its business and no longer can justify developer effort on free software. Proprietary software wins, Free Software loses.

—Kyle Rankin

Put Down the Pipe

Learn a few techniques for avoiding the pipe and making your command-line commands more efficient.

Anyone who uses the command line would acknowledge how powerful the pipe is. Because of the pipe, you can take the output from one command and feed it to another command as input. What's more, you can chain one command after another until you have exactly the output you want.

Pipes are powerful, but people also tend to overuse them. Although it's not necessarily wrong to do so, and it may not even be less efficient, it does make your commands more complicated. More important though, it also wastes keystrokes! Here I highlight a few examples where pipes are commonly used but aren't necessary.

Stop Putting Your Cat in Your Pipe

One of the most common overuses of the pipe is in conjunction with `cat`. The `cat` command concatenates multiple files from input into a single output, but it has become the overworked workhorse for piped commands. You often will find people using `cat` just to output the contents of a single file so they can feed it into a pipe. Here's the most common example:

```
cat file | grep "foo"
```

Far too often, if people want to find out whether a file contains a particular pattern, they'll `cat` the file piped into a `grep` command. This works, but `grep` can take a filename as an argument directly, so you can replace the above command with:

```
grep "foo" file
```

The next most common overuse of `cat` is when you want to sort the output from one

or more files:

```
cat file1 file2 | sort | uniq
```

Like with `grep`, `sort` supports multiple files as arguments, so you can replace the above with:

```
sort file1 file2 | uniq
```

In general, every time you find yourself catting a file into a pipe, re-examine the piped command and see whether it can accept files directly as input first either as direct arguments or as STDIN redirection. For instance, both `sort` and `grep` can accept files as arguments as you saw earlier, but if they couldn't, you could achieve the same thing with redirection:

```
sort < file1 file2 | uniq
grep "foo" < file
```

Remove Files without xargs

The `xargs` command is very powerful on the command line—in particular, when piped to from the `find` command. Often you'll use the `find` command to pick out files that have a certain criteria. Once you have identified those files, you naturally want to pipe that output to some command to operate on them. What you'll eventually discover is that commands often have upper limits on the number of arguments they can accept.

So for instance, if you wanted to perform the somewhat dangerous operation of finding and *removing* all of the files under a directory that match a certain pattern (say, all mp3s), you might be tempted to do something like this:

```
find ./ -name "*.mp3" -type f -print0 | rm -f
```

Of course, you should *never* directly pipe a `find` command to remove. First, you

should *always* pipe to `echo` to ensure that the files you are about to delete are the ones you want to delete:

```
find ./ -name "*.mp3" -type f -print0 | echo
```

If you have a lot of files that match the pattern, you'll probably get an error about the number of arguments on the command line, and this is where `xargs` normally comes in:

```
find ./ -name "*.mp3" -type f -print0 | xargs echo
find ./ -name "*.mp3" -type f -print0 | xargs rm -f
```

This is better, but if you want to delete files, you don't need to use a pipe at all. Instead, first just use the `find` command without a piped command to see what files would be deleted:

```
find ./ -name '*.mp3" -type f
```

Then take advantage of `find`'s `-delete` argument to delete them without piping to another command:

```
find ./ -name '*.mp3" -type f -delete
```

So next time you find your pinky finger stretching for the pipe key, pause for a second and think about whether you can combine two commands into one. Your efficiency and poor overworked pinky finger (whoever thought it made sense for the pinky to have the heaviest workload on a keyboard?) will thank you.

—Kyle Rankin

FOSS Project Spotlight: Mender.io, an Open-Source Over-the-Air Software Update Manager for IoT Devices

Mender is an open-source (Apache 2.0) project to address over-the-air (OTA) software update management for Linux-based IoT devices. When we researched this five years ago, there were no open-source end-to-end (device-to-server) options to manage the lifecycle of OTA updates for connected devices. Some open-source options were available, but they either had a proprietary management server, or they were client-only and required integration with another back-end server.

In short, the options available to IoT device-makers either had vendor lock-in or simply were too kludgy. Thus, we created Mender, which has two components: the runtime client integrated into the device and the management server with an intuitive user interface to manage updates at scale for large fleets.

We found in our initial research phase that many embedded systems developers created their own remote update mechanism, which usually took risky shortcuts around security and robustness. Embedded development traditionally has been a very diverse space, and the lack of technology standardization generates a lot of custom work for device-makers. Unlike web development and accepted standards, such as the LAMP stack, device-makers had to create much of their stack. This includes the fundamental capability of remote updates. And, most developers had no other choice but to build their own, given how exotic hardware and OS combinations could be for connected devices. We created a community repository called Mender Hub to

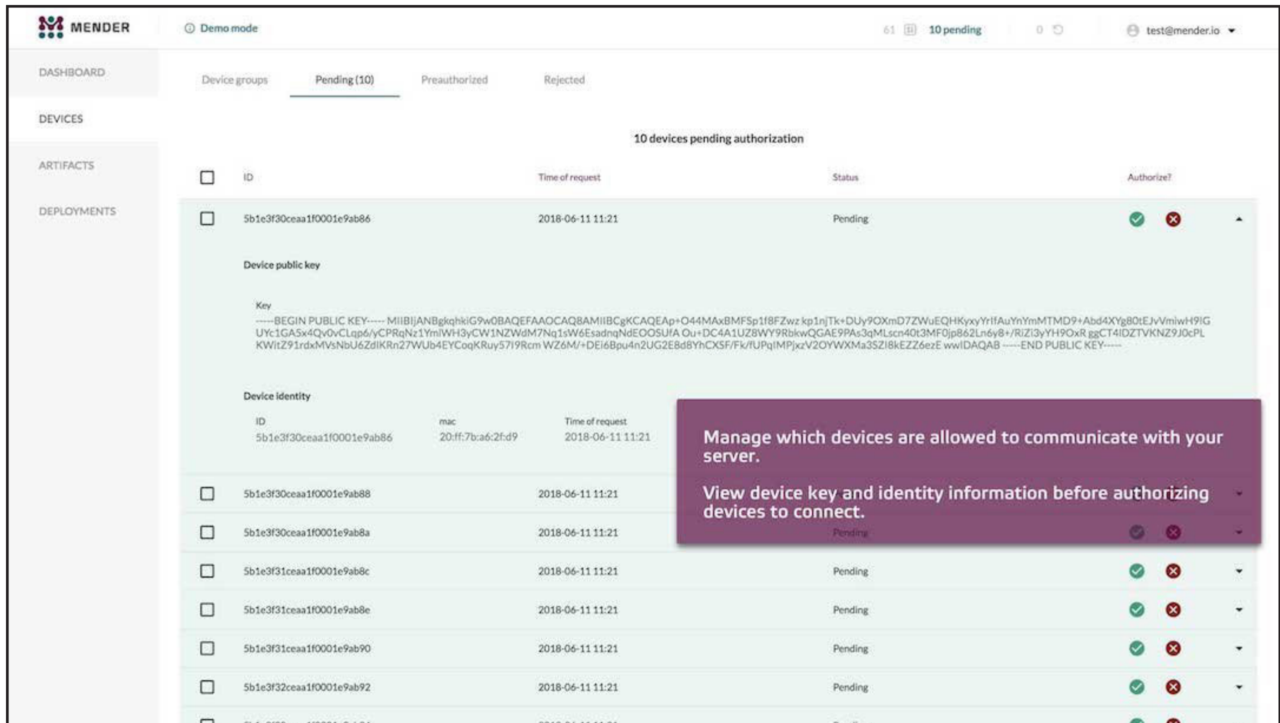


Figure 1. The Mender Server's User Interface

allow developers to create and reuse tested and validated integrations to enable OTA updates for any combination of hardware and OS.

A consequence of the growth of IoT devices is the increase of easy targets for malicious actors, evident in the proliferation of malware targeting poorly secured IoT devices. There have been an increasing number of malware attacks infecting poorly secured connected devices. The 2016 Dyn DDoS attack was one of the clearest examples of the ramifications of poorly secured IoT devices, which was executed through the Mirai malware infecting a large number of IoT devices and enslaved them into a botnet. The IoT botnet attack caused major outages across internet platforms and services, including Amazon, GitHub and Netflix.

The increasing connectivity of cars, medical devices and more is making IoT security a serious public health issue. We created Mender to help with baseline security-hardening, and security patching is fundamental. But remote updates is quite challenging and has a

lot of nuances to consider to establish a secure and robust OTA process.

There are many real-world examples of connected devices bricking or otherwise becoming unusable due to a brittle update mechanism. Devices can be bricked if an update is interrupted for any reason, including power loss on the device or poor network connectivity. Lockstate, a smart lock company recommended by Airbnb, bricked their devices after a software update and their customers were required to ship back their locks to be repaired manually. The underlying reason is as follows: "A feature update for a different set of locks accidentally included this subset of locks from a first generation 6000i WiFi lock we stopped making a year ago."

Mender has a concept of device types to make sure software can be deployed only to compatible hardware. In the situation with Lockstate, the software simply wouldn't have been able to be deployed to an incorrect version of the device, as it would have crashed at boot time, and Mender would automatically roll back to the last working version.

Fiat Chrysler also had an issue with an OTA software update causing its UConnect infotainment system to go into a reboot loop and in some cases caused the eventual draining of the vehicle's battery. Mender has adopted a dual root filesystem approach to avoid this issue, where an update would be installed in the passive rootfs partition with sanity checks to ensure it is working properly before making that partition active. In this situation, Mender's post-install scripts would have avoided this situation entirely, as Mender has automatic rollback built in.

Mender has full image updates today in order to avoid partially updated devices. The typical output of an embedded Linux CI build is a complete root filesystem, and we wanted to avoid the unmanageability of caring for individual packages. Atomic, full image updates help make deployments reproducible, as all devices will get the same version of all subcomponents. In a fleet of devices, having some with untested configurations because there was a package-based partial installation would become chaotic very quickly.

Mender's security features include requiring a secure communication channel between

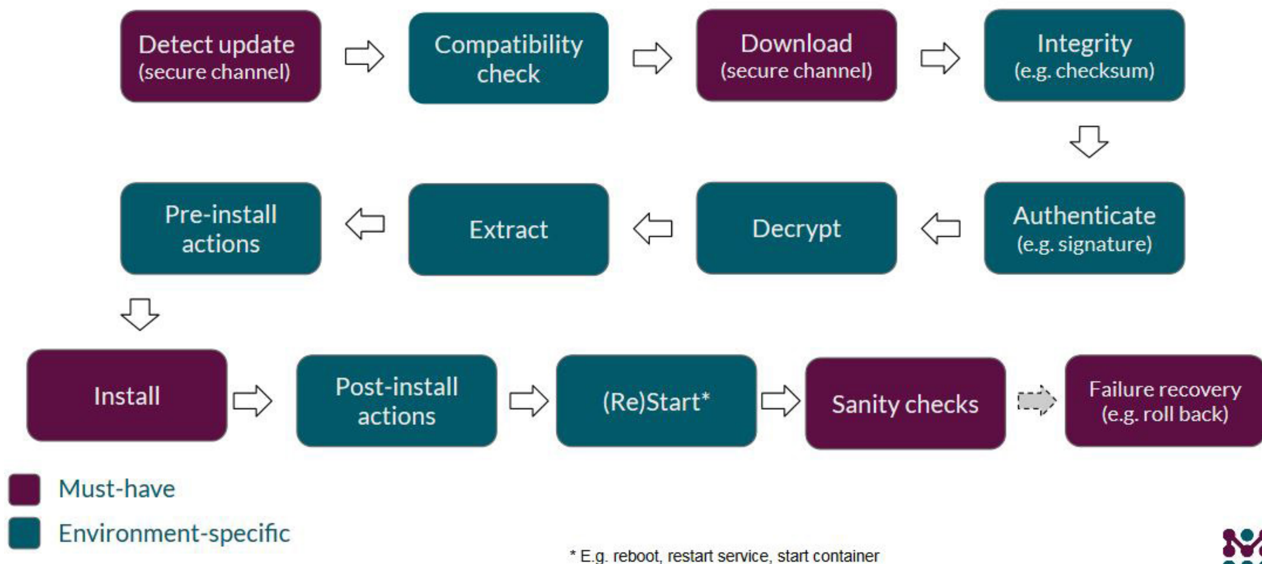


Figure 2. General IoT Software Update Workflow

the device and server with TLS. Mender also has code-signing for the verification of update artifacts, a feature that industry-leader Tesla implemented after being hacked by Tencent's Keen Security Lab, who was able to get through the vehicle's WiFi connection of a Model S and was able to reach the driving systems and manipulate the brakes while it moved.

Mender can be deployed on-premises or can be used as a service with Hosted Mender. We also have collaborated with Google to integrate Mender into Cloud IoT. Other capabilities include device groupings for controlled update rollouts and an integration to the Yocto Project, a popular build system for embedded Linux. Mender also provides out-of-the-box support for binary distributions including Debian, Raspbian and Ubuntu, and it has the Beaglebone Black and Raspberry Pi 3 as reference devices. The Mender team also is working on the ability to install updates to smaller devices as well as delta updates.

—Ralph Nguyen

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.

News Briefs

Visit LinuxJournal.com for daily news briefs.

- There's a new project called iSH that lets you run a Linux shell on an iOS device. Bleeping Computer reports that the project is available as a TestFlight beta for iOS devices, and it is based on Alpine Linux. It allows you to "transfer files, write shell scripts, or simply to use Vi to develop code or edit files". You first need to install the TestFlight app, and then you can start testing the app by visiting this page: https://testflight.apple.com/join/97i7KM8O.

- Debian is phasing out vendor-specific patches. Phoronix reports that "effective immediately these vendor-specific patches to source packages will be treated as a bug and will be unpermitted following the Debian 10 'Buster' release". See the mailing-list announcement for more information.

- Raspberry Pi 3 Model A+ is now available: "you can now get the 1.4GHz clock speed, 5GHz wireless networking and improved thermals of Raspberry Pi 3B+ in a smaller form factor, and at the smaller price of $25." You can order one here.

- The LF Deep Learning Foundation (a project of The Linux Foundation) announced the first software release of the Acumos AI Project, Athena. From the press release: "Acumos AI is a platform and open source framework that makes it easy to build, share and deploy AI applications. Acumos AI standardizes the infrastructure stack and components required to run an out-of-the-box general AI environment. This frees data scientists and model trainers to focus on their core competencies and accelerate innovation." See the full release notes here.

- Simon Long has released a new Raspbian update. This update includes a "fully hardware-accelerated version of VLC", version 3 of the Thonny Python development environment, improved desktop configuration and more. You can download the update from here.

- Uber has joined The Linux Foundation. The press release quotes Linux Foundation

Executive Director Jim Zemlin: "Uber has been active in open source for years, creating popular projects like Jaeger and Horovod that help businesses build technology at scale. We are very excited to welcome Uber to the Linux Foundation community. Their expertise will be instrumental for our projects as we continue to advance open solutions for cloud native technologies, deep learning, data visualization and other technologies that are critical to businesses today."

- Feral Interactive announced that *Shadow of the Tomb Raider* is coming to Linux in 2019. *Shadow of the Tomb Raider* is the conclusion of Laura Croft's origin story; the previous two installments are available for Linux now from Feral Interactive. You can view the *Shadow of the Tomb Raider* trailer **here**.

- **UserLAnd** is **now available on F-Droid**. With UserLAnd, you can run full Linux distributions or specific apps on top of Android, and you can install and uninstall it like a regular app—you don't need root. This version requires Android 5.0 or newer, and UserLAnd recommends that you **install the F-Droid client** to build it rather than download the APK.

- A new cybersecurity company called Darktrace has developed a tool in collaboration with the University of Cambridge that uses machine learning to detect internal security breaches. According to **FossBytes**, Darktrace created an algorithm that "recognizes new instances of unusual behavior". This technique is "based on unsupervised learning, which doesn't require humans to specify what to look for. The system works like the human body's immune system."

- France is dumping Google. *Wired* reports that to "avoid becoming a digital colony of the US or China", the French National Assembly and the French Army Ministry "**declared** that their digital devices would stop using Google as their default search engines. Instead, they will use Qwant, a French and German search engine that prides itself for not tracking its users."

- Chrome and Firefox developers plan to end support for FTP. **BleepingComputer reports** that "an upcoming change in how files stored on FTP servers are rendered

in the browser may be the first step in its ultimate removal", and also that "Google developers have advocated for the removal of FTP support in Chrome for over 4 years" due to its low usage and the additional attack surface it creates that Chrome is unable to secure properly, compared to offering the same files over an HTTPS connection.

- The CubeSat satellites that confirmed the successful landing of the Mars Insight lander on Mars contained Gumxtix's Linux-driven Overo IronStorm-Y module and Caspa VL camera. According to **Linux Gizmos**, "the Mars Cube One (MarCO) satellites are the first CubeSats to have traveled beyond low Earth orbit. They also likely represent the farthest distance a Linux computer has traveled into space."

- Microsoft is building its own Chromium browser to replace Edge on Windows 10. **The Verge reports** that "Microsoft will announce its plans for a Chromium browser as soon as this week, in an effort to improve web compatibility for Windows." The Verge article also notes that "There were signs Microsoft was about to adopt Chromium onto Windows, as the company's engineers have been working with Google to support a version of Chrome on an ARM-powered Windows operating system."

- Australia plans to give law enforcement and intelligence agencies the ability to access encrypted messages on platforms like WhatsApp, putting public safety concerns ahead of personal privacy. **Bloomberg reports** that "Amid protests from companies such as Facebook Inc. and Google, the government and main opposition struck a deal on Tuesday [December 4, 2018] that should see the legislation passed by parliament this week. Under the proposed powers, technology companies could be forced to help decrypt communications on popular messaging apps, or even build new functionality to help police access data."

Decentralized Certificate Authority and Naming

Free and open source contributors only:

handshake.org/signup

Back to Basics: Sort and Uniq

Learn the fundamentals of sorting and de-duplicating text on the command line.

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.

If you've been using the command line for a long time, it's easy to take the commands you use every day for granted. But, if you're new to the Linux command line, there are several commands that make your life easier that you may not stumble upon automatically. In this article, I cover the basics of two commands that are essential in anyone's arsenal: `sort` and `uniq`.

The `sort` command does exactly what it says: it takes text data as input and outputs sorted data. There are many scenarios on the command line when you may need to sort output, such as the output from a command that doesn't offer sorting options of its own (or the sort arguments are obscure enough that you just use the `sort` command instead). In other cases, you may have a text file full of data (perhaps generated with some other script), and you need a quick way to view it in a sorted form.

Let's start with a file named "test" that contains three lines:

```
Foo
Bar
Baz
```

`sort` can operate either on STDIN redirection, the input from a pipe, or, in the case of a file, you also can just specify the file on the command. So, the three following commands all accomplish the same thing:

```
cat test | sort
sort < test
sort test
```

And the output that you get from all of these commands is:

```
Bar
Baz
Foo
```

Sorting Numerical Output

Now, let's complicate the file by adding three more lines:

```
Foo
Bar
Baz
1. ZZZ
2. YYY
11. XXX
```

If you run one of the above `sort` commands again, this time, you'll see different output:

```
11. XXX
1. ZZZ
2. YYY
Bar
Baz
Foo
```

This is likely not the output you wanted, but it points out an important fact about `sort`. By default, it sorts alphabetically, not numerically. This means that a line that starts with "11." is sorted above a line that starts with "1.", and all of the lines that start with numbers are sorted above lines that start with letters.

To sort numerically, pass `sort` the `-n` option:

```
sort -n test

Bar
Baz
Foo
1. ZZZ
2. YYY
11. XXX
```

Find the Largest Directories on a Filesystem

Numerical sorting comes in handy for a lot of command-line output—in particular, when your command contains a tally of some kind, and you want to see the largest or smallest in the tally. For instance, if you want to find out what files are using the most space in a particular directory and you want to dig down recursively, you would run a command like this:

```
du -ckx
```

This command dives recursively into the current directory and doesn't traverse any other mountpoints inside that directory. It tallies the file sizes and then outputs each directory in the order it found them, preceded by the size of the files underneath it in kilobytes. Of course, if you're running such a command, it's probably because you want to know which directory is using the most space, and this is where `sort` comes in:

```
du -ckx | sort -n
```

Now you'll get a list of all of the directories underneath the current directory, but this time sorted by file size. If you want to get even fancier, pipe its output to the `tail` command to see the top ten. On the other hand, if you wanted the largest directories to be at the top of the output, not the bottom, you would add the `-r` option, which tells `sort` to reverse the order. So to get the top ten (well, top eight—the first line is the total, and the next line is the size of the current directory):

```
du -ckx | sort -rn | head
```

This works, but often people using the `du` command want to see sizes in more readable output than kilobytes. The `du` command offers the `-h` argument that provides "human-readable" output. So, you'll see output like `9.6G` instead of `10024764` with the `-k` option. When you pipe that human-readable output to `sort` though, you won't get the results you expect by default, as it will sort 9.6G above 9.6K, which would be above 9.6M.

The `sort` command has a `-h` option of its own, and it acts like `-n`, but it's able to parse standard human-readable numbers and sort them accordingly. So, to see the top ten largest directories in your current directory with human-readable output, you would type this:

```
du -chx | sort -rh | head
```

Removing Duplicates

The sort command isn't limited to sorting one file. You might pipe multiple files into it or list multiple files as arguments on the command line, and it will combine them all and sort them. Unfortunately though, if those files contain some of the same information, you will end up with duplicates in the sorted output.

To remove duplicates, you need the `uniq` command, which by default removes any duplicate lines that are adjacent to each other from its input and outputs

the results. So, let's say you had two files that were different lists of names:

```
cat namelist1.txt
Jones, Bob
Smith, Mary
Babbage, Walter

cat namelist2.txt
Jones, Bob
Jones, Shawn
Smith, Cathy
```

You could remove the duplicates by piping to `uniq`:

```
sort namelist1.txt namelist2.txt | uniq
Babbage, Walter
Jones, Bob
Jones, Shawn
Smith, Cathy
Smith, Mary
```

The `uniq` command has more tricks up its sleeve than this. It also can output only the duplicated lines, so you can find duplicates in a set of files quickly by adding the `-d` option:

```
sort namelist1.txt namelist2.txt | uniq -d
Jones, Bob
```

You even can have `uniq` provide a tally of how many times it has found each entry with the `-c` option:

```
sort namelist1.txt namelist2.txt | uniq -c
1 Babbage, Walter
```

```
2 Jones, Bob
1 Jones, Shawn
1 Smith, Cathy
1 Smith, Mary
```

As you can see, "Jones, Bob" occurred the most times, but if you had a lot of lines, this sort of tally might be less useful for you, as you'd like the most duplicates to bubble up to the top. Fortunately, you have the `sort` command:

```
sort namelist1.txt namelist2.txt | uniq -c | sort -nr
2 Jones, Bob
1 Smith, Mary
1 Smith, Cathy
1 Jones, Shawn
1 Babbage, Walter
```

Conclusion

I hope these cases of using `sort` and `uniq` with realistic examples show you how powerful these simple command-line tools are. Half the secret with these foundational command-line tools is to discover (and remember) they exist so that they'll be at your command the next time you run into a problem they can solve. ■

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

Python Testing with pytest: Fixtures and Coverage

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.

Improve your Python testing even more.

By Reuven Lerner

In my last two articles, I introduced pytest, a library for testing Python code (see "Testing Your Code with Python's pytest" Part I and Part II). pytest has become quite popular, in no small part because it's so easy to write tests and integrate those tests into your software development process. I've become a big fan, mostly because after years of saying I should get better about testing my software, pytest finally has made it possible.

So in this article, I review two features of pytest that I haven't had a chance to cover yet: fixtures and code coverage, which will (I hope) convince you that pytest is worth exploring and incorporating into your work.

Fixtures

When you're writing tests, you're rarely going to write just one or two. Rather, you're going to write an entire "test suite", with each test aiming to check a different path through your

code. In many cases, this means you'll have a few tests with similar characteristics, something that pytest handles with "parametrized tests".

But in other cases, things are a bit more complex. You'll want to have some objects available to all of your tests. Those objects might contain data you want to share across tests, or they might involve the network or filesystem. These are often known as "fixtures" in the testing world, and they take a variety of different forms.

In pytest, you define fixtures using a combination of the `pytest.fixture` decorator, along with a function definition. For example, say you have a file that returns a list of lines from a file, in which each line is reversed:

```
def reverse_lines(f):
   return [one_line.rstrip()[::-1] + '\n'
           for one_line in f]
```

Note that in order to avoid the newline character from being placed at the start of the line, you remove it from the string before reversing and then add a `'\n'` in each returned string. Also note that although it probably would be a good idea to use a generator expression rather than a list comprehension, I'm trying to keep things relatively simple here.

If you're going to test this function, you'll need to pass it a file-like object. In my last article, I showed how you could use a `StringIO` object for such a thing, and that remains the case. But rather than defining global variables in your test file, you can create a fixture that'll provide your test with the appropriate object at the right time.

Here's how that looks in pytest:

```
@pytest.fixture
def simple_file():
   return StringIO('\n'.join(['abc', 'def', 'ghi', 'jkl']))
```

On the face of it, this looks like a simple function—one that returns the value you'll want to use later. And in many ways, it's similar to what you'd get if you were to define a global variable by the name of "simple_file".

At the same time, fixtures are used differently from global variables. For example, let's say you want to include this fixture in one of your tests. You then can mention it in the test's parameter list. Then, inside the test, you can access the fixture by name. For example:

```
def test_reverse_lines(simple_file):
   assert reverse_lines(simple_file) == ['cba\n', 'fed\n',
 ↪'ihg\n', 'lkj\n']
```

But it gets even better. Your fixture might act like data, in that you don't invoke it with parentheses. But it's actually a function under the hood, which means it executes every time you invoke a test using that fixture. This means that the fixture, in contrast with regular-old data, can make calculations and decisions.

You also can decide how often a fixture is run. For example, as it's written now, this fixture will run once per test that mentions it. That's great in this case, when you want to compare with a list or file-like structure. But what if you want to set up an object and then use it multiple times without creating it again? You can do that by setting the fixture's "scope". For example, if you set the scope of the fixture to be "module", it'll be available throughout your tests but will execute only a single time. You can do this by passing the `scope` parameter to the `@pytest.fixture` decorator:

```
@pytest.fixture(scope='module')
def simple_file():
   return StringIO('\n'.join(['abc', 'def', 'ghi', 'jkl']))
```

I should note that giving this particular fixture "module" scope is a bad idea, since the second test will end up having a `StringIO` whose location pointer (checked with

`file.tell`) is already at the end.

These fixtures work quite differently from the traditional setup/teardown system that many other test systems use. However, the pytest people definitely have convinced me that this is a better way.

But wait—perhaps you can see where the "setup" functionality exists in these fixtures. And, where's the "teardown" functionality? The answer is both simple and elegant. If your fixture uses "yield" instead of "return", pytest understands that the post-yield code is for tearing down objects and connections. And yes, if your fixture has "module" scope, pytest will wait until all of the functions in the scope have finished executing before tearing it down.

Coverage

This is all great, but if you've ever done any testing, you know there's always the question of how thoroughly you have tested your code. After all, let's say you've written five functions, and that you've written tests for all of them. Can you be sure you've actually tested all of the possible paths through those functions?

For example, let's assume you have a very strange function, `only_odd_mul`, which multiplies only odd numbers:

```
def only_odd_mul(x, y):
   if x%2 and y%2:
       return x * y
   else:
       raise NoEvenNumbersHereException(f'{x} and/or {y}
 ↪not odd')
```

Here's a test you can run on it:

```
def test_odd_numbers():
   assert only_odd_mul(3, 5) == 15
```

Sure enough, the test passed. It works great! The software is terrific!

Oh, but wait—as you've probably noticed, that wasn't a very good job of testing it. There are ways in which the function could give a totally different result (for example, raise an exception) that the test didn't check.

Perhaps it's easy to see it in this example, but when software gets larger and more complex, it's not going to be so easy to eyeball it. That where you want to have "code coverage", checking that your tests have run all of the code.

Now, 100% code coverage doesn't mean that your code is perfect or that it lacks bugs. But it does give you a greater degree of confidence in the code and the fact that it has been run at least once.

So, how can you include code coverage with pytest? It turns out that there's a package called pytest-cov on PyPI that you can download and install. Once that's done, you can invoke pytest with the `--cov` option. If you don't say anything more than that, you'll get a coverage report for every part of the Python library that your program used, so I strongly suggest you provide an argument to `--cov`, specifying which program(s) you want to test. And, you should indicate the directory into which the report should be written. So in this case, you would say:

```
pytest --cov=mymul .
```

Once you've done this, you'll need to turn the coverage report into something human-readable. I suggest using HTML, although other output formats are available:

```
coverage html
```

This creates a directory called htmlcov. Open the index.html file in this directory using your browser, and you'll get a web-based report showing (in red) where your program still lacks coverage. Sure enough, in this case, it showed that the even-number path wasn't covered. Let's add a test to do this:

```
def test_even_numbers():
   with pytest.raises(NoEvenNumbersHereException):
       only_odd_mul(2,4)
```

And as expected, coverage has now gone up to 100%! That's definitely something to appreciate and celebrate, but it doesn't mean you've reached optimal testing. You can and should cover different mixtures of arguments and what will happen when you pass them.

Summary

If you haven't guessed from my three-part focus on pytest, I've been bowled over by the way this testing system has been designed. After years of hanging my head in shame when talking about testing, I've started to incorporate it into my code, including in my online "Weekly Python Exercise" course. If I can get into testing, so can you. And although I haven't covered everything pytest offers, you now should have a good sense of what it is and how to start using it. ■

Resources

- The pytest website is at http://pytest.org.
- An excellent book on the subject is Brian Okken's *Python testing with pytest*, published by Pragmatic Programmers. He also has many other resources, about pytest and code testing in general, at http://pythontesting.net.
- Brian's blog posts about pytest's fixtures are informative and useful to anyone wanting to get started with them.

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

Converting Decimals to Roman Numerals with Bash

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.

Decimals to Roman numerals—here we hit all the limitations of Bash shell scripting.

By Dave Taylor

My last few articles have given me a chance to relive my undergraduate computer science degree and code a Roman numeral to decimal converter. It's quite handy when you're watching old movies (when was MCMLVII anyway?), and the basic coding algorithm was reasonably straightforward. (See Dave's "Roman Numerals and Bash" and "More Roman Numerals and Bash".)

The trick with Roman numerals, however, is that it's what's known as a subtractive notation. In other words, it's not a position → value or even symbol → value notation, but a sort of hybrid. MM = 2000, and C = 100, but MMC and MCM are quite different: the former is 2100, and the latter is 1000 + (–100 + 1000) = 1900.

This means that the conversion isn't quite as simple as a mapping table, which makes it a good homework assignment for young comp-sci students!

Let's Write Some Code

In the Roman numeral to decimal conversion, a lot of the key work was done by this simple function:

```
mapit() {
   case $1 in
     I|i) value=1 ;;
     V|v) value=5 ;;
     X|x) value=10 ;;
     L|l) value=50 ;;
     C|c) value=100 ;;
     D|d) value=500 ;;
     M|m) value=1000 ;;
      * ) echo "Error: Value $1 unknown" >&2 ; exit 2 ;;
   esac
}
```

You'll need this function to proceed, but as a cascading set of conditional statements. Indeed, in its simple form, you could code a decimal to Roman numeral converter like this:

```
while [ $decvalue -gt 0 ] ; do

  if [ $decvalue -gt 1000 ] ; then
    romanvalue="$romanvalue M"
    decvalue=$(( $decvalue - 1000 ))
  elif [ $decvalue -gt 500 ] ; then
    romanvalue="$romanvalue D"
    decvalue=$(( $decvalue - 500 ))
  elif [ $decvalue -gt 100 ] ; then
    romanvalue="$romanvalue C"
    decvalue=$(( $decvalue - 100 ))
  elif [ $decvalue -gt 50 ] ; then
```

```
    romanvalue="$romanvalue L"
    decvalue=$(( $decvalue - 50 ))
  elif [ $decvalue -gt 10 ] ; then
    romanvalue="$romanvalue X"
    decvalue=$(( $decvalue - 10 ))
  elif [ $decvalue -gt 5 ] ; then
    romanvalue="$romanvalue V"
    decvalue=$(( $decvalue - 5 ))
  elif [ $decvalue -ge 1 ] ; then
    romanvalue="$romanvalue I"
    decvalue=$(( $decvalue - 1 ))
  fi

done
```

This actually works, though the results are, um, a bit clunky:

```
$ sh 2roman.sh 25
converts to roman numeral  X X I I I I I
```

Or, more overwhelming:

```
$ sh 2roman.sh 1900
converts to roman numeral  M D C C C L X X X X V I I I I I
```

I suppose there is some sort of charm to the latter, but there also are much, much better ways to simplify this. You can do all the math, but since my approach to coding is often “be lazy, get it done, move on”, let’s recognize that there are a very small number of special case numeric values:

```
900 = CM
400 = CD
90  = XC
```

```
40  = XL
9   = IX
4   = IV
```

That's really it. The notation allows only a single character subtracting from another, so you can't have CCM or IIX (the latter being correctly written as VIII), and some of the other possible two-character notations don't make sense. For example, why use VX when V is the same value?

So given that, all you really need to do is expand the `if-elseif` block to add the five possible values above, which makes for a pretty darn long code block. But before I share it, did you catch the error in the above code?

It's the other reason that the resultant Roman numerals are so darn long, actually. Let's look at just the very first conditional statement:

```
if [ $decvalue -gt 1000 ] ; then
  romanvalue="$romanvalue M"
  decvalue=$(( $decvalue - 1000 ))
```

Here's the question to ask yourself: what happens if the `$decvalue` is exactly 1000? Isn't that "M"? Yes, it is. Which means that all these conditionals are wrong; instead of being `-gt` they should be `-ge`.

With that fix, here's the big block of code:

```
while [ $decvalue -gt 0 ] ; do

  if [ $decvalue -ge 1000 ] ; then
    romanvalue="$romanvalue M"
    decvalue=$(( $decvalue - 1000 ))
  elif [ $decvalue -ge 900 ] ; then
    romanvalue="$romanvalue CM"
```

```
    decvalue=$(( $decvalue - 900 ))
  elif [ $decvalue -ge 500 ] ; then
    romanvalue="$romanvalue D"
    decvalue=$(( $decvalue - 500 ))
  elif [ $decvalue -ge 400 ] ; then
    romanvalue="$romanvalue CD"
    decvalue=$(( $decvalue - 400 ))
  elif [ $decvalue -ge 100 ] ; then
    romanvalue="$romanvalue C"
    decvalue=$(( $decvalue - 100 ))
  elif [ $decvalue -ge 90 ] ; then
    romanvalue="$romanvalue XC"
    decvalue=$(( $decvalue - 90 ))
  elif [ $decvalue -ge 50 ] ; then
    romanvalue="$romanvalue L"
    decvalue=$(( $decvalue - 50 ))
  elif [ $decvalue -ge 40 ] ; then
    romanvalue="$romanvalue XL"
    decvalue=$(( $decvalue - 40 ))
  elif [ $decvalue -ge 10 ] ; then
    romanvalue="$romanvalue X"
    decvalue=$(( $decvalue - 10 ))
  elif [ $decvalue -ge 9 ] ; then
    romanvalue="$romanvalue IX"
    decvalue=$(( $decvalue - 9 ))
  elif [ $decvalue -ge 5 ] ; then
    romanvalue="$romanvalue V"
    decvalue=$(( $decvalue - 5 ))
  elif [ $decvalue -ge 4 ] ; then
    romanvalue="$romanvalue IV"
    decvalue=$(( $decvalue - 4 ))
  elif [ $decvalue -ge 1 ] ; then
    romanvalue="$romanvalue I"
```

```
    decvalue=$(( $decvalue - 1 ))
  fi

done
```

It works (albeit with some easily removed spaces) with some basic numeric tests:

```
$ sh 2roman.sh 71
converts to roman numeral  L X X I
$ sh 2roman.sh 1997
converts to roman numeral  M CM XC V I I
$ sh 2roman.sh 666
converts to roman numeral  D C L X V I
```

The problem is, that's a long and graceless block of code, even if it does solve the problem.

Making the Code More Concise

Obviously, every block of code has the very same format:

```
elif [ $decvalue -ge VALUE ] ; then
  romanvalue="$romanvalue NOTATION-FOR-VALUE"
  decvalue=$(( $decvalue - VALUE ))
```

As highlighted, there are only two values to consider: the numeric value `VALUE` and the one or two character `NOTATION-FOR-VALUE`. As an example, `VALUE=90, NOTATION=XC`. The logical function then is:

```
SubIfValue()
{
  # if $decvalue >= $2 then add $3 to romanvalue
  # and subtract $2 from decvalue
```

```
  if [ $decvalue -ge $1 ] ; then
    romanvalue="${romanvalue}$2"
    decvalue=$(( $decvalue - $1 ))
  fi
}
```

This would produce a series of invocations like this:

```
SubIfValue  500 "D"
SubIfValue  400 "CD"
SubIfValue  100 "C"
SubIfValue   90 "XC"
```

But there's a problem with this. The loop has to iterate and subtract the largest possible value each time through; otherwise, you get very odd results.

So algorithmically, you still need to have the `if-then-elif` loop anyway:

```
if ( SubIfValue 500 "D" fails ) then
```

The problem is, that's really hard to do cleanly because you can't actually return values from a Bash shell function. Rather than be too clunky, therefore, I'm going to find a compromise on my quest to clean up the code. I can leave the function mostly the same:

```
SubValue()
{
  # add $3 to romanvalue and subtract $2 from decvalue

  romanvalue="${romanvalue}$2"
  decvalue=$(( $decvalue - $1 ))

}
```

The sequence of calls is going to look more succinct, however:

```
if [ $decvalue -ge 1000 ] ; then
  SubValue 1000 "M"
elif [ $decvalue -ge 900 ] ; then
  SubValue 900 "CM"
elif [ $decvalue -ge 500 ] ; then
  SubValue 500 "D"
elif [ $decvalue -ge 400 ] ; then
  SubValue 400 "CD"
elif [ $decvalue -ge 100 ] ; then
  SubValue 100 "C"
elif [ $decvalue -ge 90 ] ; then
  SubValue 90 "XC"
elif [ $decvalue -ge 50 ] ; then
  SubValue 50 "L"
elif [ $decvalue -ge 40 ] ; then
  SubValue 40 "XL"
elif [ $decvalue -ge 10 ] ; then
  SubValue 10 "X"
elif [ $decvalue -ge 9 ] ; then
  SubValue 9 "IX"
elif [ $decvalue -ge 5 ] ; then
  SubValue 5 "V"
elif [ $decvalue -ge 4 ] ; then
  SubValue 4 "IV"
elif [ $decvalue -ge 1 ] ; then
  SubValue 1 "I"
fi
```

And, that's the fully functional script once you wrap it in the `while;do / done` loop.

A few tests:

```
$ sh 2roman.sh 1991
converts to roman numeral MCMXCI
$ sh 2roman.sh 2222
converts to roman numeral MMCCXXII
$ sh 2roman.sh 1234
converts to roman numeral MCCXXXIV
```

So there you go. Solved. Now I can't recall why it seemed so daunting when I was in college. I will note that in a more sophisticated programming language, you could come up with a considerably shorter solution, particularly if you could utilize a two-dimensional array for number/characters pairs. But, that's not the Bash shell, so we work with what we have, right?

See you next time. Meanwhile, do you have an interesting programming puzzle? Drop me a note, and I'll have a look at it! ■

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

What's New in Kernel Development

By Zack Brown

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.

Unit Testing in the Linux Kernel

Brendan Higgins recently proposed adding unit tests to the Linux kernel, supplementing other development infrastructure such as **perf**, **autotest** and **kselftest**. The whole issue of testing is very dear to kernel developers' hearts, because Linux sits at the core of the system and often has a very strong stability/security requirement. Hosts of automated tests regularly churn through kernel source code, reporting any oddities to the mailing list.

Unit tests, Brendan said, specialize in testing standalone code snippets. It was not necessary to run a whole kernel, or even to compile the kernel source tree, in order to perform unit tests. The code to be tested could be completely extracted from the tree and tested independently. Among other benefits, this meant that dozens of unit tests could be performed in less than a second, he explained.

Giving credit where credit was due, Brendan identified **JUnit**, **Python**'s **unittest.mock** and **Googletest/Googlemock** for **C++** as the inspirations for this new **KUnit** testing idea.

Brendan also pointed out that since all code being unit-tested

is standalone and has no dependencies, this meant the tests also were deterministic. Unlike on a running Linux system, where any number of pieces of the running system might be responsible for a given problem, unit tests would identify problem code with repeatable certainty.

Daniel Vetter replied extremely enthusiastically to Brendan's work. In particular, he said, "Having proper and standardized infrastructure for kernel unit tests sounds terrific. In other words: I want." He added that he and some others already had been working on a much more specialized set of unit tests for the **Direct Rendering Manager** (DRM) driver. Brendan's approach, he said, would be much more convenient than his own more localized efforts.

Dan Williams was also very excited about Brendan's work, and he said he had been doing a half-way job of unit tests on the **libnvdimm** (non-volatile device) project code. He felt Brendan's work was much more general-purpose, and he wanted to convert his own tests to use KUnit.

Tim Bird replied to Brendan's initial email as well, saying he thought unit tests could be useful, but he wanted to make sure the behaviors were correct. In particular, he wanted clarification on just how it was possible to test standalone code. If the code were to be compiled independently, would it then run on the local system? What if the local system had a different hardware architecture from the system for which the code was intended? Also, who would maintain unit tests, and where would the tests live, within the source tree? Would they clutter up the directory being tested, or would they live far away in a special directory reserved for test code? And finally, would test code be easier to write than the code being tested? In other words, could new developers cut their teeth on a project by writing test code, as a gateway to helping work on a given driver or subsystem? Or would unit tests have to be written by people who had total expertise in the area already?

Brendan attempted to address each of those issues in turn. To start, he confirmed that the test code was indeed extracted and compiled on the local system. Eventually, he said, each test would compile into its own completely independent test binary,

although for the moment, they were all lumped together into a single **user-mode-linux** (UML) binary.

In terms of cross-compiling test code for other architectures, Brendan felt this would be hard to maintain and had decided not to support it. Tests would run locally and would not depend on architecture-specific characteristics.

In terms of where the unit tests would live, Brendan said they would be in the same directory as the code being tested. So every directory would have its own set of unit tests readily available and visible. The same person maintaining the code being tested would maintain the tests themselves. The unit tests, essentially, would become an additional element of every project. That maintainer would then presumably require that all patches to that driver or subsystem pass all the unit tests before they could be accepted into the tree.

In terms of who was qualified to write unit tests for a given project, Brendan explained:

> In order to write a unit test, the person who writes the test must understand what the code they are testing is supposed to do. To some extent that will probably require someone with some expertise to ensure that the test makes sense, and indeed a change that breaks a test should be accompanied by an update to the test. On the other hand, I think understanding what pre-existing code does and is supposed to do is much easier than writing new code from scratch, and probably doesn't require too much expertise.

Brendan added that unit tests would probably reduce, rather than increase, a maintainer's workload. In spite of representing more code overall:

> Code with unit tests is usually cleaner, the tests tell me exactly what the code is supposed to do, and I can run the tests (or ideally have an automated service run the tests) that tell me that the code actually does what the tests say it should. Even when it comes to writing code, I find that writing code with unit tests ends up saving me time.

Overall, Brendan was very pleased by all the positive interest, and said he planned

to do additional releases to address the various technical suggestions that came up during the course of discussion. No voices really were raised in opposition to any of Brendan's ideas. It appears that unit tests may soon become a standard part of many drivers and subsystems.

Ditching Out-of-Date Documentation Infrastructure

Long ago, the Linux kernel started using **00-Index** files to list the contents of each documentation directory. This was intended to explain what each of those files documented. **Henrik Austad** recently pointed out that those files have been out of date for a very long time and were probably not used by anyone anymore. This is nothing new. Henrik said in his post that this had been discussed already for years, "and they have since then grown further out of date, so perhaps it is time to just throw them out."

He counted hundreds of instances where the 00-index file was out of date or not present when it should have been. He posted a patch to rip them all unceremoniously out of the kernel.

Joe Perches was very pleased with this. He pointed out that .rst files (the kernel's native documentation format) had largely taken over the original purpose of those 00-index files. He said the oo-index files were even misleading by now.

Jonathan Corbet was more reserved. He felt Henrik should distribute the patch among a wider audience and see if it got any resistance. He added:

> I've not yet decided whether I think this is a good idea or not. We certainly don't need those files for stuff that's in the RST doctree, that's what the index.rst files are for. But I suspect some people might complain about losing them for the rest of the content. I do get patches from people updating them, so some folks do indeed look at them.

Henrik told Jonathan he was happy to update the 00-index files if that would be preferable. But he didn't want to do that if the right answer was just to get rid of them.

Meanwhile, **Josh Triplett** saw no reason to keep the 00-index files around at all. He remarked, "I was *briefly* tempted, reading through the files, to suggest ensuring that the one-line descriptions from the 00-INDEX files end up in the documents themselves, but the more I think about it, I don't think even that is worth anyone's time to do."

Paul Moore also voiced his support for removing the 00-index files, at least the ones for **NetLabel**, which was his area of interest.

The discussion ended there. It's nice that even for apparently obvious patches, the developers still take the time to consider various perspectives and try to retain any value from the old thing to the new. It's especially nice to see this sort of attention given to documentation patches, which tend to get left out in the cold when it comes to coding projects.

Non-Child Process Exit Notification Support

Daniel Colascione submitted some code to support processes knowing when others have terminated. Normally a process can tell when its own child processes have ended, but not unrelated processes, or at least not trivially. Daniel's patch created a new file in the /proc directory entry for each process a file called "exithand" that is readable by any other process. If the target process is still running, attempts to `read()` its exithand file will simply block, forcing the querying process to wait. When the target process ends, the `read()` operation will complete, and the querying process will thereby know that the target process has ended.

It may not be immediately obvious why such a thing would be useful. After all, non-child processes are by definition unrelated. Why would the kernel want to support them keeping tabs on each other? Daniel gave a concrete example, saying:

> Android's lmkd kills processes in order to free memory in response to various memory pressure signals. It's desirable to wait until a killed process actually exits before moving on (if needed) to killing the next process. Since the processes that lmkd kills are not lmkd's children, lmkd currently lacks a way to wait for a process to actually die after

> being sent SIGKILL.

Daniel explained that on **Android**, the **lmkd** process currently would simply keep checking the proc directory for the existence of each process it tried to kill. By implementing this new interface, instead of continually polling the process, lmkd could simply wait until the `read()` operation completed, thus saving the CPU cycles needed for continuous polling.

And more generally, Daniel said in a later email:

> I want to get polling loops out of the system. Polling loops are bad for wakeup attribution, bad for power, bad for priority inheritance, and bad for latency. There's no right answer to the question "How long should I wait before checking $CONDITION again?". If we can have an explicit waitqueue interface to something, we should. Besides, PID polling is vulnerable to PID reuse, whereas this mechanism (just like anything based on struct pid) is immune to it.

Joel Fernandes suggested, as an alternative, using **ptrace()** to get the process exit notifications, instead of creating a whole new file under /proc. Daniel explained:

> Only one process can ptrace a given process at a time, so I don't like ptrace as a mechanism for anything except debugging. Relying on ptrace for exit notification would interfere with things like debuggers and crash dump collection systems. Besides, ptrace can do too much (like read and write process memory) and so requires very strong privileges not necessary for this mechanism. Besides: ptrace's interface is complicated and relies on repeated calls to various wait functions, whereas the interface in this patch is simple enough to use from the shell.

The issue of **PID** (process ID) reuse came up again, because it wasn't clear to everyone that a whole new file in the /proc directory was the best way to solve the problem. As **David Laight** said, Linux used a reference counter on all PIDs, so that any reuse could be seen. He figured the /proc directory should include some way to expose that reference count.

Other operating system kernels have other ways of trying to avoid PIT reuse or at least mitigate its downsides. As Joel explained:

> If you look at the NetBSD pid allocator you'll see that it uses the low pid bits to index an array and the high bits as a sequence number. The array slots are also reused LIFO, so you always need a significant number of pid allocate/free before a number is reused. The non-sequential allocation also makes it significantly more difficult to predict when a pid will be reused. The table size is doubled when it gets nearly full.

But to this, Daniel replied:

> NetBSD is still just papering over the problem. The real issue is that the whole PID-based process API model is unsafe, and a clever PID allocator doesn't address the fundamental race condition. As long as PID reuse is possible at all, there's a potential race condition, and correctness depends on hope. The only way you could address the PID race problem while not changing the Unix process API is by making pid_t ridiculously wide so that it never wraps around.

Elsewhere, **Aleksa Sarai** was still unconvinced that that a whole new file in the /proc directory would be a good thing, if there were a way to avoid it. Aleksa understood that Daniel wanted to avoid continuous polling, but felt there were still workable alternatives. For example, Aleksa said, "When you open /proc/$pid, you already have a handle for the underlying process, and you can already poll to check whether the process has died (fstatat fails for instance). What if we just used an inotify event to tell userspace that the process has died—to avoid userspace doing a poll loop?"

Daniel replied that Aleksa's solution was far more complicated than Daniel's. He said that **inotify** and related APIs were:

> ...intended for broad monitoring of system activity, not for waiting for some specific event. They require a substantial amount of setup code, and since both are event-streaming APIs with buffers that can overflow, both need some logic for userspace to detect buffer overrun and fall back to explicit scanning if that happens. They're also

> optional part of the kernel.

Daniel went on:

> Given that we *can*, cheaply, provide a clean and consistent API to userspace, why would we instead want to inflict some exotic and hard-to-use interface on userspace instead? Asking that userspace poll on a directory file descriptor and, when poll returns, check by looking for certain errors (we'd have to spec which ones) from fstatat is awkward. /proc/pid is a directory. In what other context does the kernel ask userspace to use a directory this way?

The debate went on, with no resolution on the mailing list. Daniel continued to insist that his approach was simpler than any of the proposed alternatives, and he also felt it was in keeping with the spirit of UNIX itself. At one point, he explained:

> The basic unix data access model is that a userspace application wants information (e.g., next bunch of bytes in a file, next packet from a socket, next signal from a signal FD, etc.), and tells the kernel so by making a system call on a file descriptor. Ordinarily, the kernel returns to userspace with the requested information when it's available, potentially after blocking until the information is available. Sometimes userspace doesn't want to block, so it adds O_NONBLOCK to the open file mode, and in this mode, the kernel can tell the userspace requestor "try again later", but the source of truth is still that ordinarily-blocking system call. How does userspace know when to try again in the "try again later" case? By using select/poll/epoll/whatever, which suggests a good time for that "try again later" retry, but is not dispositive about it, since that ordinarily-blocking system call is still the sole source of truth, and that poll is allowed to report spurious readabilty. This model works fine and has a ton of mental and technical infrastructure built around it. It's the one the system uses for almost every bit of information useful to an application.

The opposition to Daniel's patch seems to emanate from the desire to avoid adding new files to /proc. There's a real risk of /proc, and other kernel interfaces, growing bloated, overly complex and unmaintainable over time. **Linus Torvalds** and other top contributors want to avoid this, especially since it is very difficult to remove

interfaces once they are implemented. Once user software starts to rely on a given interface, there's a great reluctance in Linux to break that software. One reason for this is that not all software is open source, and older closed-source tools may not be maintained, and thus may not have the option to adapt to any new interface. A change in something they rely on may mean the software simply can't be used with newer kernels. The kernel developers want to avoid that situation if at all possible.

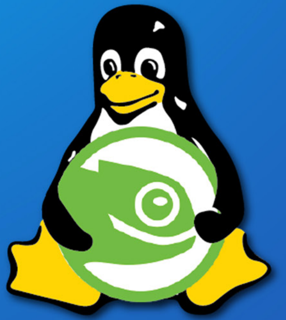
It's unclear whether Daniel's patch will go into the tree in its current form, given the opposition. It may be that user code—the Android OS in this case—for now will have to continue to use other, more complicated ways of knowing when processes have died. ■

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.

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

DEEP DIVE

DISTRIBUTIONS



The State of Desktop Linux 2019

A snapshot of the current state of Desktop Linux at the start of 2019—with comparison charts and a roundtable Q&A with the leaders of three top Linux distributions.

By Bryan Lunduke

I've never been able to stay in one place for long—at least in terms of which Linux distribution I call home. In my time as a self-identified "Linux Person", I've bounced around between a number of truly excellent ones. In my early days, I picked up boxed copies of S.u.S.E. (back before they made the U uppercase and dropped the dots entirely) and Red Hat Linux (before Fedora was a thing) from store shelves at various software outlets.

Debian, Ubuntu, Fedora, openSUSE—I spent a good amount of time living in the biggest distributions around (and many others). All of them were fantastic. Truly stellar. Yet, each had their own quirks and peculiarities.

Side note: remember when we used to buy Operating Systems—and even most software—in actual boxes, with actual physical media and actual printed manuals? I still have big printed manuals for a few early Linux versions, which, back then, were necessary for getting just about everything working (from X11 to networking and sound). Heck, sometimes simply getting a successful boot required a few trips through those heavy manuals. Ah, those were the days.

As I bounced from distro to distro, I developed a strong attachment to just about all of them, learning, as I went, to appreciate each for what it was. Just the same, when asked which distribution I recommend to others, my brain begins to melt down. Offering any single recommendation feels simply inadequate.

Choosing which one to call home, even if simply on a secondary PC, is a deeply personal choice.

Maybe you have an aging desktop computer with limited RAM and an older, but still absolutely functional, CPU. You're going to need something light on system resources that runs on 32-bit processors.

Or, perhaps you work with a wide variety of hardware architectures and need a single operating system that works well on all of them—and standardizing on a single Linux distribution would make it easier for you to administer and update all of them. But what options even are available?

To help make this process a bit easier, I've put together a handy set of charts and graphs to let you quickly glance and find the one that fits your needs (Figures 1 and 2).

But, let's be honest, knowing that a particular system meets your hardware needs (and preferences) simply is not enough. What is the community like? What's in store for the future of this new system you are investing in? Do the ideals of its leadership match up with your own?

In the interests of helping to answer those questions, I sat down with the leaders of three of the most prominent Linux distros of the day:

- Chris Lamb: Debian Project Leader
- Daniel Fore: elementary Founder
- Matthew Miller: Fedora Project Leader

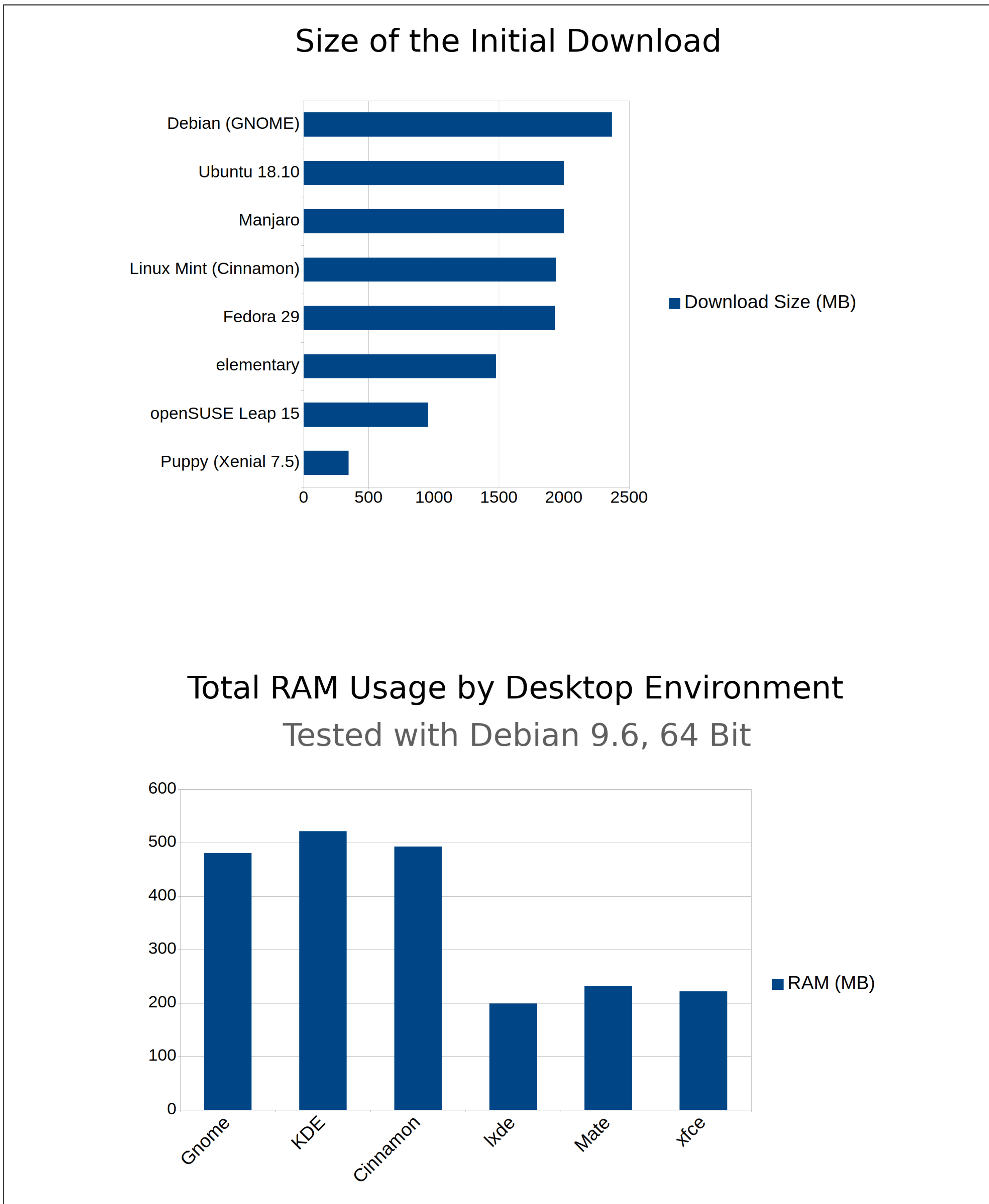
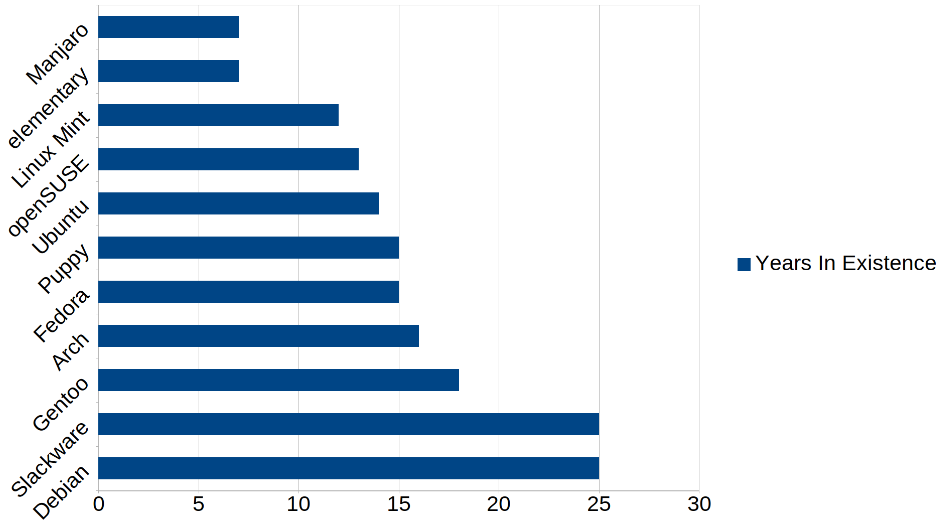


Figure 1. Distribution Comparison Chart I



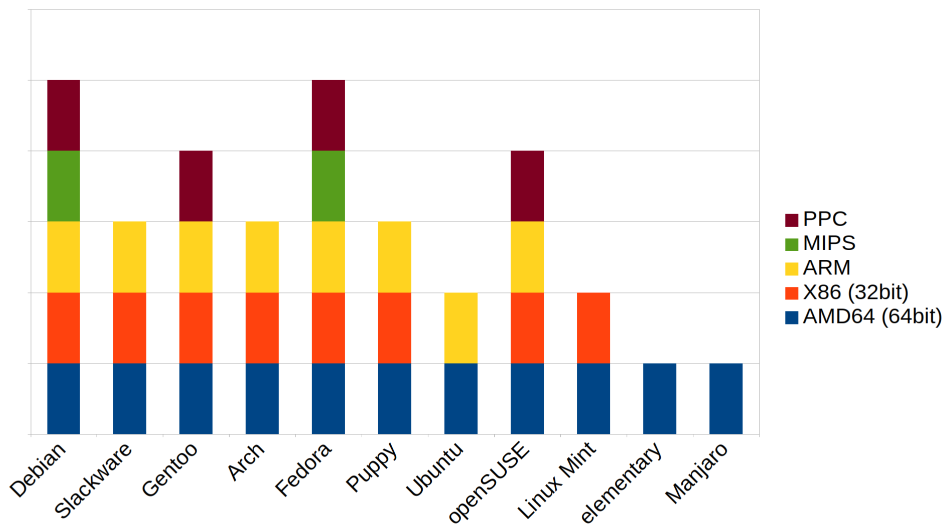


Figure 2. Distribution Comparison Chart II

Each of these systems is unique, respected and brings something truly valuable to the world.

I asked all three leaders the exact same questions—and gave each the chance to respond to each other. The topics are all over the place and designed to help show the similarities and differences between the distributions, both in terms of goals and culture.

Note that the Fedora project leader, Matthew Miller, was having an unusually busy time (both for work and personally), but he still made time to answer as many questions as he could. That, right there, is what I call dedication.

Bryan (*LJ*):
Introduce your Linux distribution (the short, elevator-pitch version—just a few sentences) and your role within it.

Daniel (elementary):
elementary is focused on growing the market for open-source software and chipping away at the share of our closed-source competitors. We believe in providing a great user experience for both new users and pro users, and putting a strong emphasis on security and privacy. We build elementary OS: a consumer-focused operating system for desktops and notebooks.

My role at elementary is as Founder and CEO. I work with our various teams (like design, development, web and translation teams) to put together a cohesive vision, product roadmap and ensure that we're following an ethical path to sustainable funding.

Chris (Debian):
The Debian Project, which celebrated its 25th birthday this year, is one of the oldest and largest GNU/Linux distributions and is run on an entirely volunteer basis.

Not only does it have stellar reputation for stability and technical excellence, it

has a unwavering philosophical stance on free software (i.e., it comes with no proprietary software pre-installed and the main repository is only free software). As it underpins countless derivative distributions, such as Ubuntu, et al., it is uniquely poised and able to improve the Free Software world as a whole.

The Debian Project Leader (DPL) is a curious beast. Far from being a BDFL—the DPL has no authoritative or deciding say in technical matters—the project leader is elected every year to a heady mix of figurehead, spokesperson and focus/contact point, but the DPL is also responsible for the quotidian business of keeping the project moving with respect to reducing bureaucracy and smoothing any and all roadblocks to Debian Developers' productivity.

Matthew (Fedora):
The Fedora distribution brings all of the innovation of thousands of upstream projects and hundreds of thousands of upstream developers together into a polished operating system for users, with releases on a six-month cadence. We're a community project tied together through the shared project mission and through the "four Fs" of our foundations: Freedom, Friends, Features and First. Something like 3,000 people contribute directly to Fedora in any given year, with a core active group of around 400 people participating in any given week.

We just celebrated the 15th anniversary of our first release, but our history goes back even further than that to Red Hat Linux. I'm the Fedora Project Leader, a role funded by Red Hat—paying people to work on the project is the largest way Red Hat acts as a sponsor. It's not a dictatorial role; mostly, I collect good ideas and write short persuasive essays about them. Leadership responsibility is shared with the Fedora Council, which includes both funded roles, members selected by parts of the community and at-large elected representatives.

Bryan (*LJ*):
With introductions out of the way, let's start with this (perhaps deceptively) simple question:

How many Linux distributions should there be? And why?

Daniel (elementary):
As long as there are a set of users who aren't getting their needs met by existing options, there's a purpose for any number of distros to exist. Some come and some go, and many are very very niche, but that's okay. I think there's a lot of people who are obsessed with trying to have some dominant player take a total monopoly, but in every other market category, it's immediately apparent how silly that idea is. You wouldn't want a single clothing manufacturer or a single restaurant chain or a single internet provider (wink hint nudge) to have total market dominance. Diversity and choice in the marketplace is good for customers, and I think it's no different when it comes to operating systems.

Matthew (Fedora):
[Responding to Daniel] Yes, I agree exactly. That said, creating an entirely from scratch distro is a lot of work, and a lot of it not very interesting work. If you've got something innovative at the how-we-put-the-OS-together level (like CoreOS), there's room for that, but if you're focused higher up the stack, like a new desktop environment or something else around user experience, it makes the most sense to make a derivative of one of the big community-powered distros. There's a lot of boring hard work, and it makes sense to reuse rather than carry those same rocks to the top of a slightly different hill.

In Fedora, we're aiming to make custom distro creation as easy as possible. We have "spins", which are basically mini custom distros. This is stuff like the Python Classroom Lab or Fedora Jam (which is focused on musicians). We have a framework for making those *within* the Fedora project—I'm all about encouraging bigger, broader sharing and collaboration in Fedora. But if you want to work outside the project—say, you really have different ideas on free and open-source vs. proprietary software—we have Fedora Remixes that let you do that.

Chris (Debian):
The competing choice of distributions is often cited as a reason preventing Linux

from becoming mainstream as it robs the movement of a consistent and focused marketing push.

However, philosophical objections against monopolistic behaviour granted, the diversity and freedom that this bazaar of distributions affords is, in my view, paradoxically exactly why it has succeeded.

That people are free—but more important, feel free—to create a new distribution as a means to try experimental or outlandish approaches to perceived problems is surely sufficient justification for some degree of proliferation or even duplication of effort.

In this capacity, Debian's technical excellence, flexibility and deliberate lack of a top-down direction has resulted in it becoming the base underpinning countless derivatives, clearly and evidently able to provide the ingredients to build one's "own" distribution, often without overt credit.

Matthew wrote: "if you want to work outside the project—say, you really have different ideas on free and open source vs. proprietary software—we have Fedora Remixes that let you do that."

Given that, I would be curious to learn how you protect your reputation if you encourage, or people otherwise use your infrastructure, tools and possibly even your name to create and distribute works that are antithetical to the cause of software and user freedom?

Bryan (*LJ*):
Thinking about it from a slightly different angle—how many distros would be TOO many distros?

Daniel (elementary):
More than the market can sustain I guess? The thing about Linux is that it powers all kinds of stuff. So even for one non-technical person, they could still end up running a handful of distros for their notebook, their router, their phone someday, IoT devices,

etc. So the number of distros that could exist sustainably could easily be in the hundreds or thousands, I think.

Chris (Debian):
If I may be so bold as to interpret this more widely, whilst it might look like we have "too many" distributions, I fear this might be misunderstanding the reasons why people are creating these newer offerings in the first place.

Apart from the aforementioned distros created for technical experimentation, someone spinning up their own distribution might be (subconsciously!) doing it for the delight and satisfaction in building something themselves and having their name attached to it—something entirely reasonable and justifiable IMHO.

To then read this creation through a lens of not being ideal for new users or even some silly "Linux worldwide domination" metric could therefore even be missing the point and some of the sheer delight of free software to begin with.

Besides, the "market" for distributions seems to be doing a pretty good job of correcting itself.

Bryan (*LJ*):
Okay, since you guys brought it up, let's talk about world domination.

How much of what you do (and what your teams do) is influenced by a desire to increase marketshare (either of your distribution specifically or desktop Linux in general)?

Daniel (elementary):
When we first started out, elementary OS was something we made for fun out of a desire to see something exist that we felt didn't yet. But as the company, and our user base, has grown, it's become more clear that our mission must be about getting open-source software in the hands of more people. As of now, our estimated userbase is somewhere in the hundreds of thousands with more than 75% of downloads coming

from users of closed-source operating systems, so I think we're making good progress toward that goal. Making the company mission about reaching out to people directly has shaped the way we monetize, develop products, market and more, by ensuring we always put users' needs and experiences first.

Chris (Debian):
I think it would be fair to say that "increasing market share" is not an overt nor overly explicit priority for Debian.

In our 25-year history, Debian has found that if we just continue to do good work, then good things will follow.

That is not to say that other approaches can't work or are harmful, but chasing potentially chimeric concepts such as "market share" can very easily lead to negative outcomes in the long run.

Matthew (Fedora):
A project's user base is directly tied to its ability to have an effect in the world. If we were just doing cool stuff but no one used it, it really wouldn't matter much. And, no one really comes into working on a distro without having been a user first. So I guess to answer the question directly for me at least, it's pretty much all of it—even things that are not immediately related are about helping keep our community healthy and growing in the long term.

Bryan (*LJ*):
The three of you represent distros that are "funded" in very different ways. Fedora being sponsored (more or less) by Red Hat, elementary being its own company and Debian being, well, Debian.

I would love to hear your thoughts around funding the work that goes into building a distribution. Is there a "right" or "ideal" way to fund that work (either from an ethical perspective or a purely practical one)?

Chris (Debian):
Clearly, melding "corporate interests" with the interests of a community distribution can be fraught with issues.

I am always interested to hear how other distros separate influence and power particularly in terms of increasing transparency using tools such as Councils with community representation, etc. Indeed, this question of "optics" is often highly under-appreciated; it is simply not enough to be honest, you must be seen to be honest too.

Unfortunately, whilst I would love to be able to say that Debian is by-definition free (!) of all such problems by not having a "big sister" company sitting next to it, we have a long history of conversations regarding the role of money in funding contributors.

For example, is it appropriate to fund developers to do work that might not not be done otherwise? And if it is paid for, isn't this simply a feedback loop that effectively ensures that this work will cease to within the remit of volunteers. There are no easy answers and we have no firm consensus, alas.

Daniel (elementary):
I'm not sure that there's a single right way, but I think we have the opinion that there are some wrong ways. The biggest questions we're always trying to ask about funding are where it's coming from and what it's incentivizing. We've taken a hard stance that advertising income is not in the interest of our users. When companies make their income from advertising, they tend to have to make compromises to display advertising content instead of the things their users actually want to see, and oftentimes are they incentivized to invade their users' privacy in order to target ads more effectively. We've also chosen to avoid big enterprise markets like server and IoT, because we believe that since companies will naturally be incentivized to work on products that turn a profit, that making that our business model would result in things like the recent Red Hat acquisition or in killing products that users love, like Ubuntu's Unity.

Instead, we focus on things like individual sales of software directly to our users,

bug bounties, Patreon, etc. We believe that doing business directly with our users incentivizes the company to focus on features and products that are in the benefit of those paying customers. Whenever a discussion comes up about how elementary is funded, we always make a point to evaluate if that funding incentivizes outcomes that are ethical and in the favor of our users.

Regarding paying developers, I think elementary is a little different here. We believe that people writing open-source software should be able to make a living doing it. We owe a lot to our volunteer community, and the current product could not be possible without their hard work, but we also have to recognize that there's a significant portion of work that would never get done unless someone is being paid to do it. There are important tasks that are difficult or menial, and expecting someone to volunteer their time to them after their full work day is a big ask, especially if the people knowledgeable in these domains would have to take time away from their families or personal lives to do so. Many tasks are also just more suited to sustained work and require the dedicated attention of a single person for several weeks or months instead of some attention from multiple people over the span of years. So I think we're pretty firmly in the camp that not only is it important for some work to be paid, but the eventual goal should be that anyone writing open-source code should be able to get paid for their contributions.

Chris (Debian):
Daniel wrote: "So I think we're pretty firmly in the camp that not only is it important for some work to be paid, but the eventual goal should be that anyone writing open-source code should be able to get paid."

Do you worry that you could be creating a two-tier community with this approach?

Not only in terms of hard influence (eg. if I'm paid, I'm likely to be able to simply spend longer on my approach) but moreover in terms of "soft" influence during discussions or by putting off so-called "drive-thru" contributions? Do you do anything to prevent the appearance of this?

Matthew (Fedora):
Chris wrote: "Do you worry that you could be creating a two-tier community with this approach?"

Yeah, this is a big challenge for us. We have many people who are paid by Red Hat to work on Fedora either full time or as part of their job, and that gives a freedom to just be *around* a lot more, which pretty much directly translates to influence. Right now, many of the community-elected positions in Fedora leadership are filled by Red Hatters, because they're people the community knows and trusts. It takes a lot of time and effort to build up that visibility when you have a different day job. But there's some important nuances here too, because many of these Red Hatters aren't actually paid to work on Fedora at all—they're doing it just like anyone else who loves the project.

Daniel (elementary):
Chris wrote: "Do you worry that you could be creating a two-tier community with this approach?"

It's possible, but I'm not sure that we've measured anything to this effect. I think you might be right that employees at elementary can have more influence just as a byproduct of having more time to participate in more discussions, but I wouldn't say that volunteers' opinions are discounted in any way or that they're underrepresented when it comes to major technical decisions. I think it's more that we can direct labor *after* design and architecture decisions have been discussed. As an example, we recently had decided to make the switch from CMake to Meson. This was a group discussion primarily led by volunteers, but the actual implementation was then largely carried out by employees.

Chris (Debian):
Daniel wrote: "Do you worry that you could be creating a two-tier community with this approach? ... It's possible, but I'm not sure that we've measured anything to this effect."

I think it might be another one of those situations where the optics in play is perhaps as important as the reality. Do you do anything to prevent the appearance of any bias?

Not sure how best to frame it hypothetically, but if I turned up to your project tomorrow and learned that some developers were paid for their work (however fairly integrated in practice), that would perhaps put me off investing my energy.

Bryan (*LJ*):
What do you see as the single biggest challenge currently facing both your specific project—and desktop Linux in general?

Daniel (elementary):
Third-party apps! Our operating systems are valuable to people only if they can use them to complete the tasks that they care about. Today, that increasingly means using proprietary services that tie in to closed-source and non-native apps that often have major usability and accessibility problems. Even major open-source apps like Firefox don't adhere to free desktop standards like shipping a .desktop file or take advantage of new cross-desktop metadata standards like AppStream. If we want to stay relevant for desktop users, we need to encourage the development of native open-source apps and invest in non-proprietary cloud services and social networks. The next set of industry-disrupting apps (like DropBox, Sketch, Slack, etc.) need to be open source and Linux-first.

Chris (Debian):
Third-party apps/stores are perhaps the biggest challenge facing all distributions within the medium- to long-term, but whilst I would concede there are cultural issues in play here, I believe they have some element of being technical challenges or at least having some technical ameliorations.

More difficult, however, is that our current paradigms of what constitutes software freedom are becoming difficult to square with the increased usage of cloud services. In the years ahead we may need to revise our perspectives, ideas and possibly even our definitions of what constitutes free software.

There will be a time when the FLOSS community will have to cease the casual mocking of "cloud" and acknowledge the reality that it is, regardless of one's view of it, here to stay.

Matthew (Fedora):
For desktop Linux, on the technical side, I'm worried about hardware enablement—not just the work dealing with driver compatibility and proprietary hardware, but more fundamentally, just being locked out. We've just seen Apple come out with hardware locked so Linux won't even boot—even with signed kernels. We're going to see more of that, and more tablets and tablet-keyboard combos with similar locked, proprietary operating systems.

A bigger worry I have is with bringing the next generation to open source—a lot of Fedora core contributors have been with the project since it started 15 years ago, which on the one hand is awesome, but also, we need to make sure that we're not going to end up with no new energy. When I was a kid, I got into computers through programming BASIC on an Apple][. I could see commercial software and easily imagine myself making the same kind of thing. Even the fanciest games on offer—I could see the pixels and could use PEEK and POKE to make those beeps and boops. But now, with kids getting into computers via *Fortnite* or whatever, that's not something one can just sit down and make an approximation of as a middle-school kid. That's discouraging and makes a bigger hill to climb.

This is one reason I'm excited about Fedora IoT—you can use Linux and open source at a tinkerer's level to make something that actually has an effect on the world around you, and actually probably a lot *better* than a lot of off-the-shelf IoT stuff.

Bryan (*LJ*):
Where do you see your distribution in five years? What will be its place be in the broader Linux and computing world?

Chris (Debian):
Debian naturally faces some challenges in the years ahead, but I sincerely believe that the Project remains as healthy as ever.

We are remarkably cherished and uniquely poised to improve the free software ecosystem as a whole. Moreover, our stellar reputation for technical excellence, stability and software freedom remains highly respected where losing this would surely be the beginning of the end for Debian.

Daniel (elementary):
Our short-term goals are mostly about growing our third-party app ecosystem and improving our platform. We're investing a lot of time into online accounts integration and working with other organizations, like GNOME, to make our libraries and tooling more compelling. Sandboxed packaging and Wayland will give us the tools to help keep our users' data private and to keep their operating system stable and secure. We're also working with OEMs to make elementary OS more shippable and to give users a way to get an open-source operating system when they buy a new computer. Part of that work is the new installer that we're collaborating with System76 to develop. Overall, I'd say that we're going to continue to make it easier to switch away from closed-source operating systems, and we're working on increasing collaborative efforts to do that.

Bryan (*LJ*):
When you go to a FOSS or Linux conference and see folks using Mac and Windows PCs, what's your reaction? Is it a good thing or a bad thing when developers of Linux software primarily use another platform?

Chris (Debian):
Rushing to label this as a "good" or "bad" thing can make it easy to miss the underlying and more interesting lessons we can learn here.

Clearly, if everyone was using a Linux-based operating system, that would be a better state of affairs, but if we are overly quick to dismiss the usage of Mac

systems as "bad", then we can often fail to understand why people have chosen to adopt the trade-offs of these platforms in the first place.

By not demonstrating sufficient empathy for such users as well as newcomers or those without our experience, we alienate potential users and contributors and tragically fail to communicate our true message. Basically, we can be our own worst enemy sometimes.

Daniel (elementary):
Within elementary, we strongly believe in dogfood, but I think when we see someone at a conference using a closed-source operating system, it's a learning opportunity. Instead of being upset about it or blaming them, we should be asking why we haven't been able to make a conversion. We need to identify if the problem is a missing product, feature, or just with outreach and then address that.

Bryan (*LJ*):
How often do you interact with the leaders of other distributions? And is that the right amount?

Chris (Debian):
Whilst there are a few meta-community discussion groups around, they tend to have a wider focus, so yes, I think we could probably talk a little more, even just as a support group or a place to rant!

More seriously though, this conversation itself has been fairly insightful, and I've learned a few things that I think I "should" have known already, hinting that we could be doing a better job here.

Daniel (elementary):
With other distros, not too often. I think we're a bit more active with our partners, upstreams and downstreams. It's always interesting to hear about how someone else tackles a problem, so I would be interested in interacting more with others, but in a lot of cases, I think there are philosophical or technical differences that

mean our solutions might not be relevant for other distros.

Bryan (*LJ*):
Is there value in the major distributions standardizing on package management systems? Should that be done? Can that be done?

Chris (Debian):
I think I would prefer to see effort go toward consistent philosophical outlooks and messaging on third-party apps and related issues before I saw energy being invested into having a single package management format.

I mean, is this really the thing that is holding us all back? I would grant there is some duplication of effort, but I'm not sure it is the most egregious example and—as you suggest—it is not even really technically feasible or is at least subject to severe diminishing returns.

Daniel (elementary):
For users, there's a lot of value in being able to sideload cross-platform, closed-source apps that they rely on. But outside of this use case, I'm not sure that packaging is much more than an implementation detail as far as our users are concerned. I do think though that developers can benefit from having more examples and more documentation available, and the packaging formats can benefit from having a diverse set of implementations. Having something like Flatpak or Snap become as well accepted as SystemD would probably be good in the long run, but our users probably never noticed when we switched from Upstart, and they probably won't notice when we switch from Debian packages.

Bryan (*LJ*):
Big thanks to Daniel, Matthew and Chris for taking time out to answer questions and engage in this discussion with each other. Seeing the leadership of such excellent projects talking together about the things they differ on—and the things they align on completely—warms my little heart. ■

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**.

Resources

- Debian Project
- Debian's Unwavering Philosophical Stance on Free Software
- Debian's 25th Birthday
- Benevolent Dictator for Life (Wikipedia)
- Debian Project Leader Elections 2017
- Debian Project Leader Elections 2018
- Bits from the DPL (October 2018)
- Get Fedora
- Fedora's Mission and Foundations
- Celebrate 15 Years of Fedora
- elementary OS
- Publish on AppCenter

Linux and the Multiverse

A look at the rich diversity of Linux distributions.

By Marcel Gagné

What do Linux distributions and the Nobel Prize-winning work by Saul Perlmutter, Brian P. Schmidt and Adam G. Riess have in common? Well, Linux was originally the hobby project of one Linus Torvalds back in 1991 when he lived in Helsinki, Finland. Perlmutter, on the other hand, worked on the Supernova Cosmology Project at the Lawrence Berkeley National Laboratory and the University of California in Berkeley. Schmidt was part of the High-z Supernova Search Team at Australian National University, and Riess was also on the High-z Supernova Search Team but worked out of Johns Hopkins University and Space Telescope Science Institute in Baltimore.

You see where I'm going with this? The supernova team won the 2011 Nobel Prize for physics for "the discovery of the accelerating expansion of the Universe through observations of distant supernovae". In short, they discovered that the universe is not only expanding, as Edwin Hubble observed back in 1929 when he noticed that everything seemed to be moving away from us, but that the expansion was accelerating. This is a big deal, because everyone assumed that gravity would eventually do its dirty work and slow the whole expanding mess down. That turns out not to be the case.

So what's causing this anti-gravity force? Dark energy, for which the team actually came up with a number, a number which, as it turns out, is super tiny and its source, unknown. Later work, based on these observations, suggests that string theory might hold the answer, while others point to the Higgs Field, long theorized but

only recently confirmed. Spoiler alert: nobody knows for sure, but if you follow this whole thing down the proverbial rabbit hole, you wind up concluding that there are countless universes in addition to our own—what we now refer to as the multiverse.

Just as the possibility exists for countless universes, so does the possibility exist for countless Linux distributions, When Linus chose to open the code for his new kernel, he unknowingly set in motion a kind of "distribution Big Bang", where the original code, combined with other open-source projects, began stretching out into the furthest reaches of the internet, where those combinations could spawn other versions of what eventually would form what we now think of as distributions. Just as matter from the early universe coalesced into dust clouds and then into stars that through their eventual cataclysmic destruction in supernovae would spawn the heavier elements that would, in time, create our own solar system with our planetary home, the Earth, so too did this early code evolve to create the rich diversity of Linux distributions.

We tend to think of our Earth as being a pretty mundane place, which is why we invented the Star Trek Federation, Agartha, Middle Earth, Narnia, Dune and Westeros. The same is true for Linux distributions. Sure, we all could be running a single version of Linux, like Red Hat or Ubuntu, but that would go against the very laws of the, ahem, multiverse. Just as we aren't always aware of the many universes that exist, you may not be aware as to just how many Linux distributions there are. Today, I'm going to give you a sample. Best of all, although it has proven to be insanely difficult to travel to any universe but our own, trying a different Linux distribution is as easy as downloading an ISO and rebooting. Welcome to the Linuverse.

To the Moon!

This one seems like a great place to start, because, well, it's out of this world. You can get Lunar Linux (Figure 1) from lunar-linux.org where you'll find images compressed using the xz format. Consequently, your first step is to `unxz` the file:

```
unxz lunar-1.7.0-x86_64.iso.xz
```

Lunar boots using a classic text screen, and that's something I never want to see disappear from the Linuverse. The installation is fascinating, because it's somewhat reminiscent of Slackware, with its blue text screen, but each item invites you to take "one step forward". If you mess up, you can take one step back. You'll be asked to select language and keyboard, and to partition your disk. There's a variety of tools for the latter, but I picked fdisk, just because. After jumping through a few hoops and answering a variety of questions, you'll need to install a kernel (which will be visible from the boot screen) and finally reboot.

Yes, this is a classic "Take chances, make mistakes and get messy" distribution, as Ms. Frizzle from the Magic School Bus would say. And, messy you will get. Once you boot and log in to root, the first thing you'll do is build your X environment from scratch. That's right:

```
lin XOrg7
```

That command will start a dialog asking you to choose the various installation

```
root@lunar ~ # lin XOrg7
xproto: Install & use optional (not installed) module xmlto
          Purpose: for documentation ? [n]
freetype2: Use optional (installed) module zlib
          Purpose: for zlib compression support, no=use internal zlib instead of system ? [y]
freetype2: Use optional (installed) module bzip2
          Purpose: for support of bzip2 compressed fonts ? [y]
bzip2: Enable large file support? [y]
freetype2: Install & use optional (not installed) module libpng
          Purpose: for support of png compressed OpenType embeded bitmaps ? [n] y
mkfontscale: Use optional (installed) module bzip2
          Purpose: for bzip2 compressed bitmap font support ? [y]
libXmu: Enable IPv6? [y]
libXt: Enable XKB support? [y]
libSM: Enable IPv6? [y]
libSM: Use optional (installed) module util-linux
          Purpose: Build with libuuid support for client IDs? ? [y]
libxcb: Install & use optional (not installed) module libXdmcp
          Purpose: for X Display Manager Control Protocol support (recommended) ? [n]
libX11: Install & use optional (not installed) module xf86bigfontproto
          Purpose: for XF86BigFont extension ? [n]
xinit: Install & use optional (not installed) module xterm
          Purpose: for a very basic terminal (not needed by modern Desktop Environments) ? [n] y
_
```

Figure 1. Lunar Linux, where anything you want, including X, you need to build.

components (Figure 1). Stay with me. This is actually fun.

Once you build X, you'll probably want a desktop with that distribution, so like X, you'll need to build that too. I happen to like KDE, so I decided to build that:

```
lin kde4
```

Substitute `xfce4` for `kde4` if you want XFCE or `gnome2` if you want GNOME for your desktop environment. If you're bored with the simplicity of `apt-get` or `yum` for installing packages, and you long for those early days of compiling everything, Lunar is for you. A word of warning though: you may find that the Lunar cache isn't always fresh, and `lin` might have trouble finding the odd package. If that happens, visit the lunar download page on the web, find your package, and download it manually. Oh, you'll definitely want to install "links" before you do that:

```
lin links
```

The repository is here. If you find yourself having to download a package, say "libpng", as I did, you'll need to install it by referencing your local directory (for example, /tmp):

```
lin -f /tmp -w 1.6.35 libpng
```

What that means is find the source bundle in /tmp where you want (the `-w` flag) version 1.6.35 of the package. It's a strange bit of nostalgic fun, and I spent far more time on this than I would have thought possible.

Paldo Linux

Maybe you don't want to start right from scratch, and you would fancy a desktop environment to work from, in which case you might want to try Paldo (Figure 2), which stands for "pure adaptable Linux distribution". Paldo is a strange beast in that it looks a lot like a typical GNOME-based desktop system, but the philosophy is one I hadn't run into before. Think of it as a source-and-binary hybrid where packages

are built and binaries installed, but with all source and development files installed by default. Packages are not split, so you get everything related to that package. You can make local changes and use local "differential" repositories for maximum flexibility.

Another thing that sets Paldo apart from other distributions is its choice of package manager, UPKG. To install LibreOffice, for example, you would use the following command:

```
upkg-install libreoffice
```

The `upkg` approach, along with Paldo's stated goal of building a "just works" system,

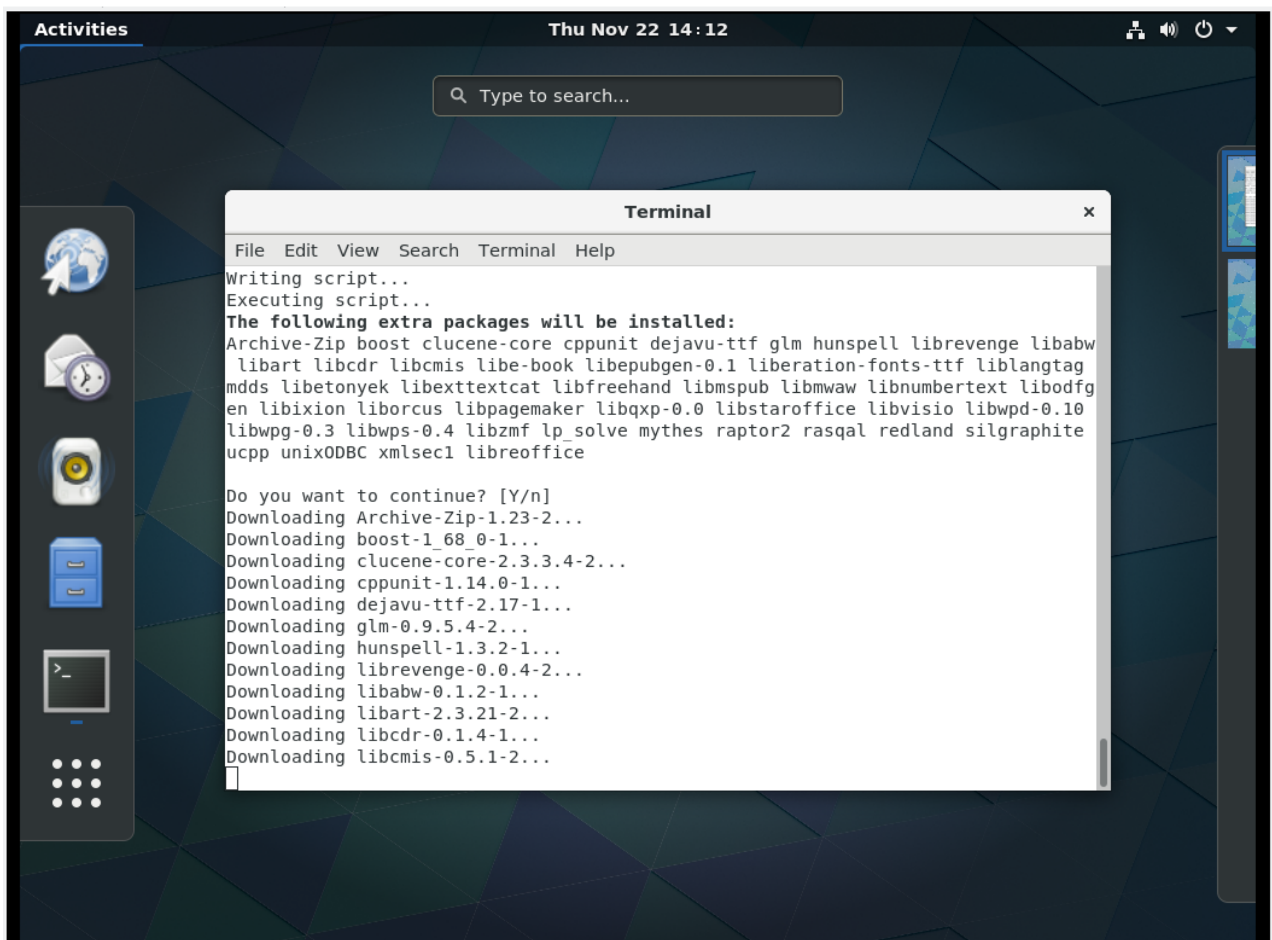


Figure 2. Using `upkg` to Install LibreOffice in Paldo

does come at a price. It does not provide an everything package that you might have come to expect, choosing instead what it considers to be the right program for the right task. For instance, when it comes to desktop environment, you have one choice: GNOME. Also, the source/binary approach means some packages will download source and build locally, which can take some getting used to and is definitely not recommended for anyone lacking in patience.

In another part of the Linuverse, Linux plays alongside *Pokemon*, *Fullmetal Alchemist* and *Yu-Gi-Oh!*, and the distribution they run is Linux Mangaka (Figure 3), a Manga-focused distribution based on Ubuntu LTS releases. The current version, Cho

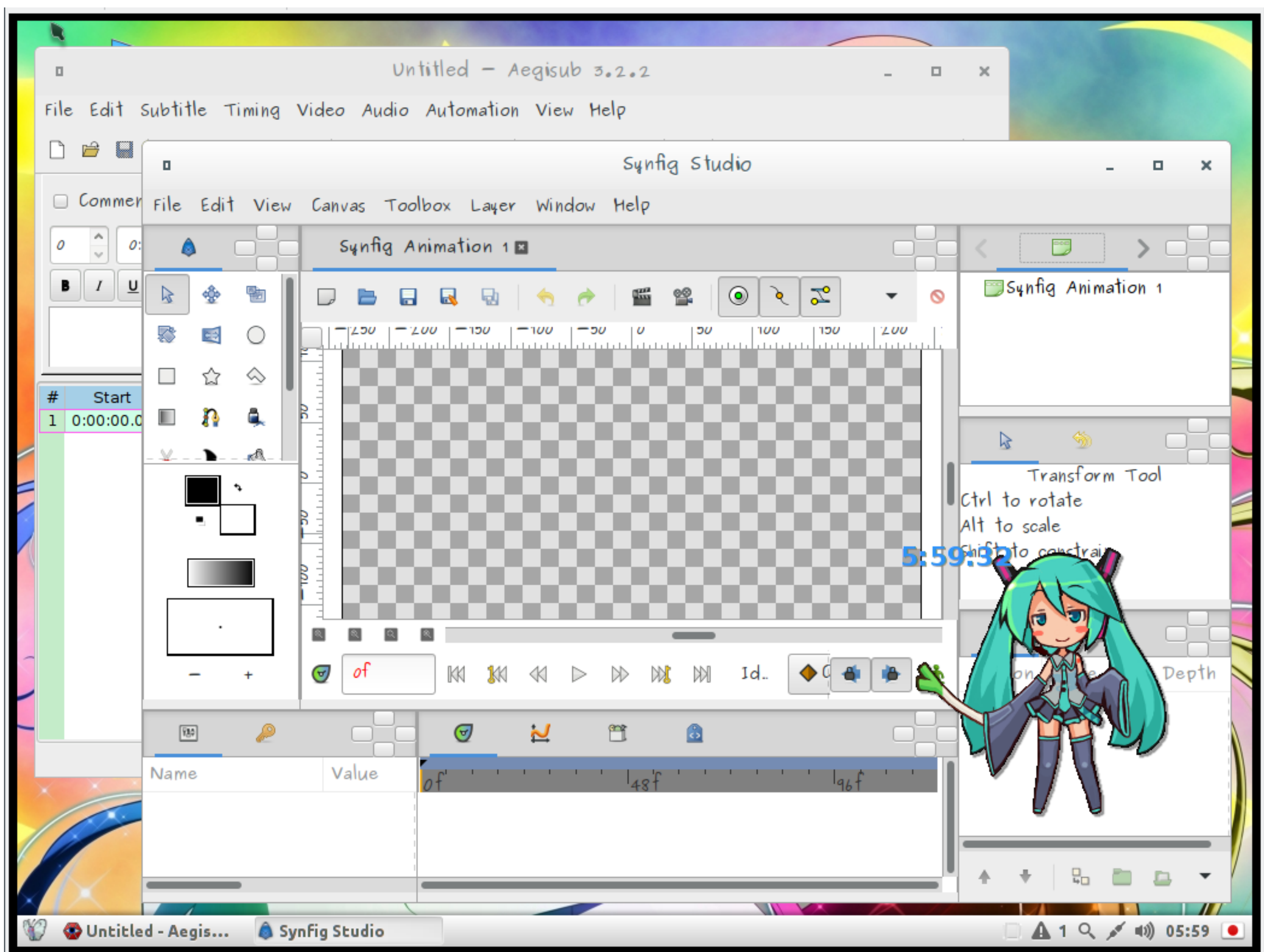


Figure 3. Linux Mangaka contains tools for Manga creators and enthusiasts.

(Japanese for "butterfly"), runs Ubuntu 16.04 LTS with AIO (Japanese for "love"), which likely will be out by the time this article is published.

Aside from the colorful Manga-styled desktop artwork, there's a serious distribution under the surface that's more than straight Ubuntu. In addition to things like Comix, a comic-book-reader app, there's Synfig, a powerful tool for creating 2D animation, and Aegisub, an application to allow fans to create their own subtitles for foreign language videos.

Figure 4. The First Issue of *Ubunchu!*, a Manga for Ubuntu Users

One more thing before I move on. In the Mangaka part of the Linuverse, the number-one favorite Manga is called *Ubunchu!*, the ongoing story of three young students in a school computer lab, all facing the challenges we encounter every day. It's truly inspiring stuff. We're lucky in that this Ubuntu-themed Manga, of which there are now seven issues, is available for download and your reading pleasure.

Now, it's time to get serious again. With infinite possibilities, you get some fascinating and highly specialized creations. For instance, eucalyptus leaves are far from nutritious and, for most animals, are actually poisonous. That doesn't stop Australia's Koala from living on a diet of these plants. With millions of years of evolution and some serious specialization, you can produce some pretty amazing creatures.

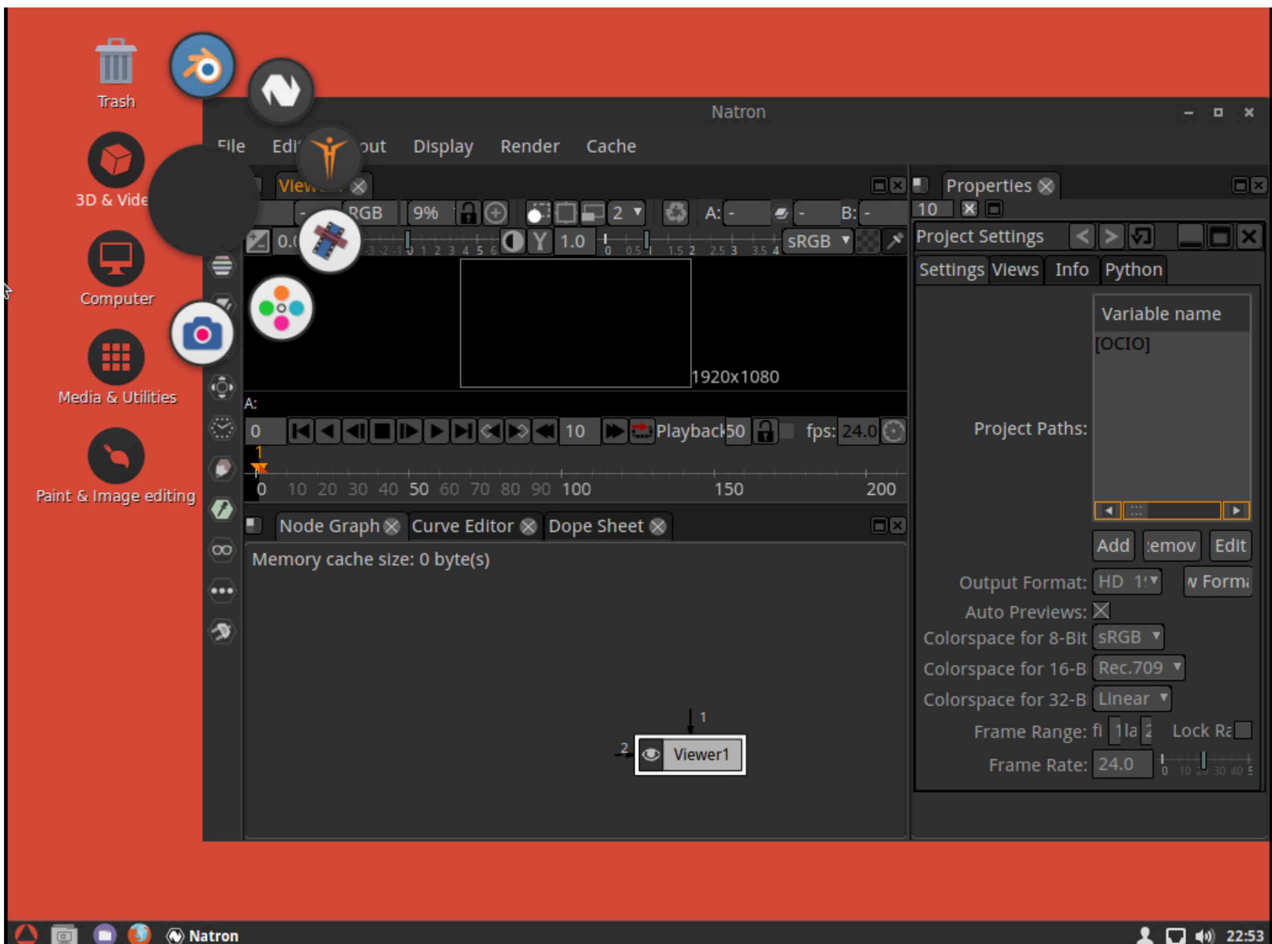


Figure 5. Iro OS, a Distribution for the Visual and Film Arts

In the Linuverse, specialization can spawn fun distributions, like Hannah Montana Linux (seriously, go look it up; I'm not helping you with this one), but that also makes it possible to create distributions that focus on a collection of niche or industry-specific tools. Take Iro OS (Figure 5), a Linux distribution built specifically for animators, video and film producers, artists and others who work in the visual arts. It's also the first time I've seen the GNOME Pie menus, so that was a pretty cool discovery in and of itself. You can see the "3D and Video" menu opened up in the screenshot shown in Figure 5.

The collection of applications is rich but industry-specific. There's Blender for 3D modeling and rendering, Inkscape for vector graphics, MakeHuman to create and render realistic anatomically correct human figures, Natron for node-based compositing, Kdenlive for video editing, Synfig (which I mentioned earlier), and a few more familiar apps like GIMP, Krita and others. It's a well thought out set of tools, and the distribution looks super slick in red and black.

Way, Way Outside!

With an infinite number of universes, there may be no end to the extent of "weird and wonderful" out there. Why should it be any different with the world of Linux and open source? I want to wrap up this exploration with a couple other distributions that, although not Linux-based, are fully open source, and give you a glimpse into that infinite collection of weird and wonderful.

Behold, my friends, an operating system for everypony—not everybody, but *everypony*. This is PonyOS (Figure 6). The developer of PonyOS swears that although it feels Linux-ish, the kernel is not Linux (or Hurd or Mimix, etc.), but rather it's something cooked up by the mind or minds behind this odd distribution. PonyOS has its own package manager, runs on a variety of hardware, albeit small, and it easily runs inside 512MB. It's so small, in fact, that I ran it entirely in memory and from the command line, like this:

```
qemu-system-i386 -m 512M -enable-kvm -soundhw ac97 ponyos.iso
```

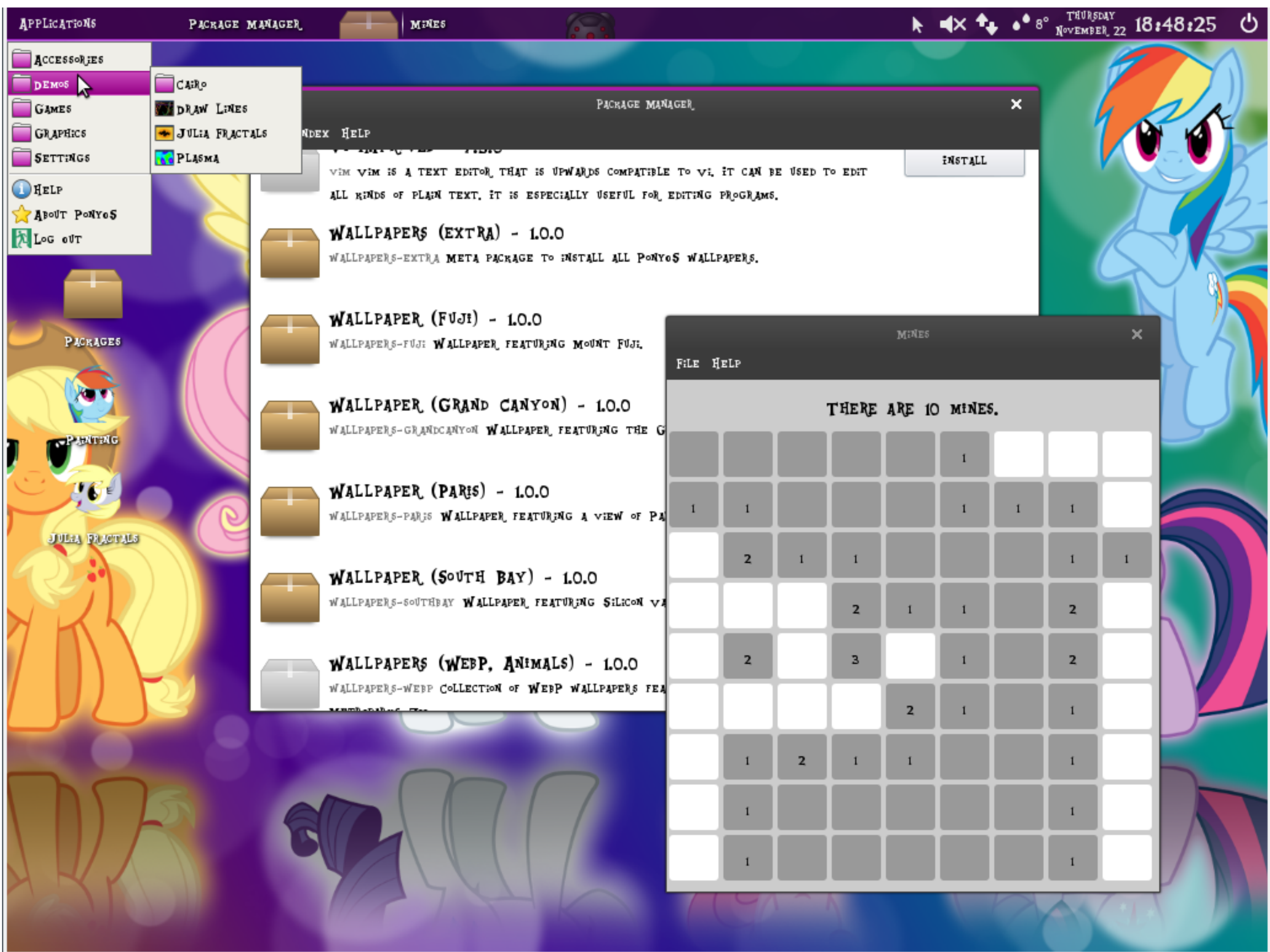


Figure 6. PonyOS, the OS for Everypony

You can open a terminal window and play some games, or open up vim and get creative—maybe cook up your own wacky distribution.

Reacting to Windows

Way back when, before Microsoft loved Linux (it says so right there on its website), there was a kind of friendly enemy relationship between the open world of Linux and the tightly closed world of Microsoft. Let's just say we didn't get along. Part of the Linux master plan was to get everyone to leave that terrible virus-prone Windows system and move to the land of rock-solid freedom that was the Linux desktop. In an effort to pull people away, we created all sorts of Windows-like desktops and even advertised them as such. Developers also built Wine, an open-source compatibility

layer that would allow some Windows software to run on top of Linux.

Some people in the Open Source world, however, chose to go a lot further, and they created the altogether unmistakably Windows-like ReactOS (Figure 7).

Perhaps the most interesting thing about ReactOS is its resilience over time. When it first appeared, it was seen as more of a joke than anything else. Yet, years later, it's still going strong, with a large number of developers continuing to improve it. These days, it feels like an almost perfect copy of Windows NT, right down to the installation process. Most interesting is the number of apps that you can install from

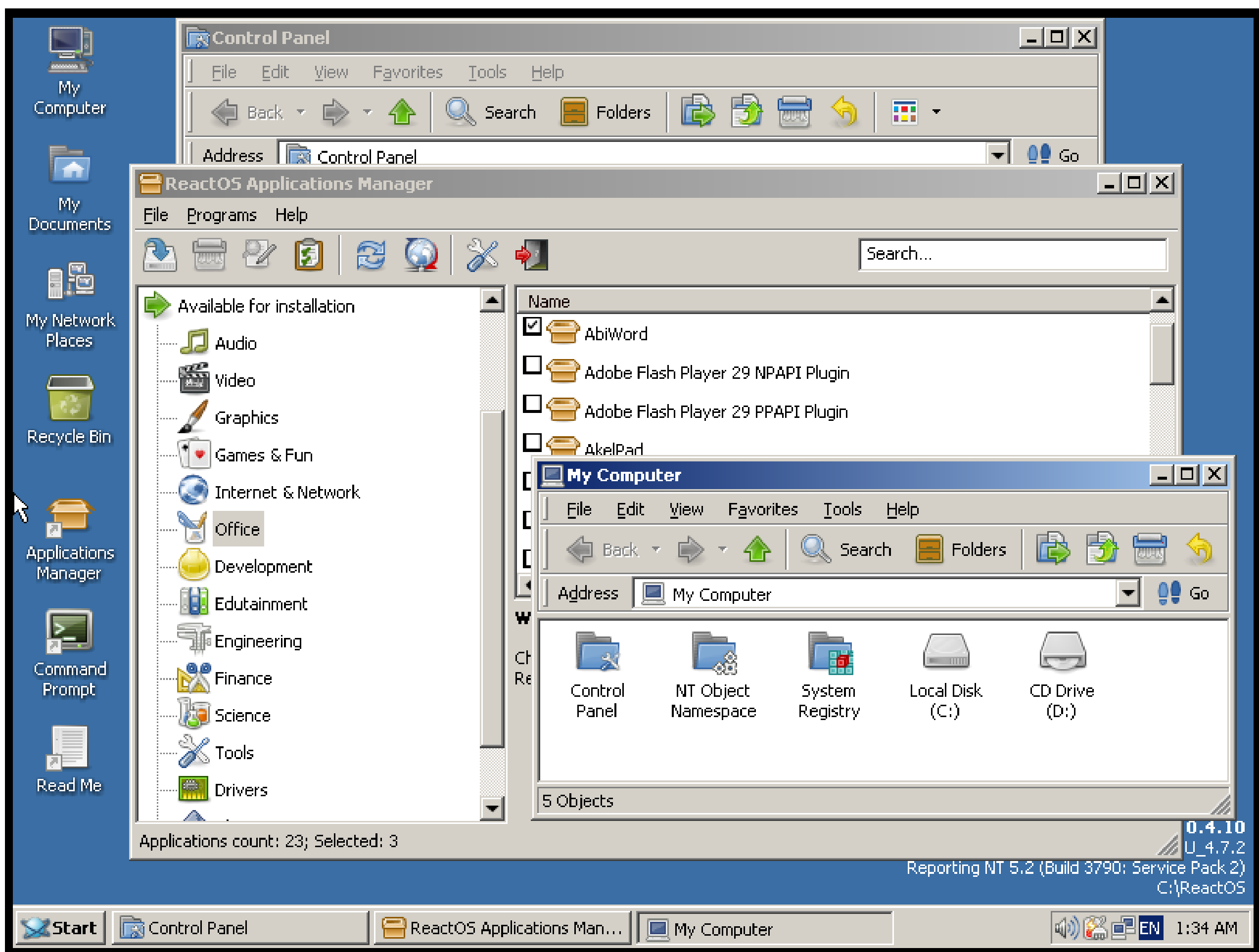


Figure 7. ReactOS even has a number of apps you can install from inside the OS.

the Add/Remove Programs section of the Control Panel, including open-source programs like LibreOffice and GnuCash.

The Linuverse may not be totally infinite, but the potential for the new, the interesting, the strange and the downright bizarre means that your Linux and open-source journey has only just begun.

Now, let's see what other weirdness I can find. ■

Marcel Gagné is Writer and Free Thinker at Large. The Cooking With Linux guy. Ruggedly handsome! Science, Linux and technology geek. Occasionally opinionated. Always confused. Loves wine, food, music and the occasional single malt Scotch.

Resources

- Lunar Linux
- Paldo Linux
- Linux Mangaka
- Iro OS
- PonyOS
- ReactOS
- Hannah Montana Linux

Build a Custom Minimal Linux Distribution from Source, Part II

Follow along with this step-by-step guide to creating your own distribution.

By Petros Koutoupis

In an **article in the June 2018 issue of *LJ***, I introduced a basic recipe for building your own minimal Linux-based distribution from source code packages. The guide started with the compilation of a cross-compiler toolchain that ran on your host system. Using that cross-compiler, I explained how to build a generic x86-64 target image, and the Linux Journal Operating System (LJOS) was born.

This guide builds on what you learned from **Part I**, so if you haven't already, be sure to go through those original steps up to the point where you are about to package the target image for distribution.

Gathering the Packages

To follow along, you'll need the following:

- busybox-1.28.3.tar.bz2 (the same package used in Part I).

Glossary

Here's a quick review the terminology from the first part of this series:

- **Host**: the *host* signifies the very machine on which you'll be doing the vast majority of work, including cross-compiling and installing the target image.

- **Target**: the *target* is the final cross-compiled operating system that you'll be building from source packages. You'll build it using the cross-compiler on the *host* machine.

- **Cross-Compiler**: you'll be building and using a *cross-compiler* to create the *target* image on the *host* machine. A cross-compiler is built to run on a host machine, but it's used to compile for an architecture or microprocessor that isn't compatible with the target machine.

- clfs-embedded-bootscripts-1.0-pre5.tar.bz2 (the same package used in Part I).

- Dropbear-2018.76.tar.bz2.

- Iana-etc-2.30.tar.bz2.

- netplug-1.2.9.2.tar.bz2.

- sysstat-12.1.1.tar.gz.

Note: I ended up rebuilding this distribution with the 4.19.1 Linux kernel. If you want to do the same, be sure to install the development package of the OpenSSL libraries on your host machine or else the build will fail. On distributions like Debian or Ubuntu, this package is named libssl-dev.

Fixing Some Boot-Time Errors

After following along with Part I, you will have noticed that during boot time, a couple errors are generated (Figure 1).

Let's clear out some of those errors. The first one relates to a script not included in BusyBox: `usbdisk_link`. For the purpose of this exercise (and because it isn't important for this example), remove the references to both `usbdisk_link` and `ide_link` in the ${LJOS}/etc/mdev.conf file. Refer to the following `diff` output to see what I mean (focus closely on the lines that begin with both `sd` and `hd`):

```
--- mdev.conf.orig      2018-11-10 18:10:14.561278714 +0000
+++ mdev.conf   2018-11-10 18:11:07.277759662 +0000
@@ -26,8 +26,8 @@ ptmx    root:tty 0666
 # ram.*
 ram([0-9]*)     root:disk 0660 >rd/%1
 loop([0-9]+)    root:disk 0660 >loop/%1
```

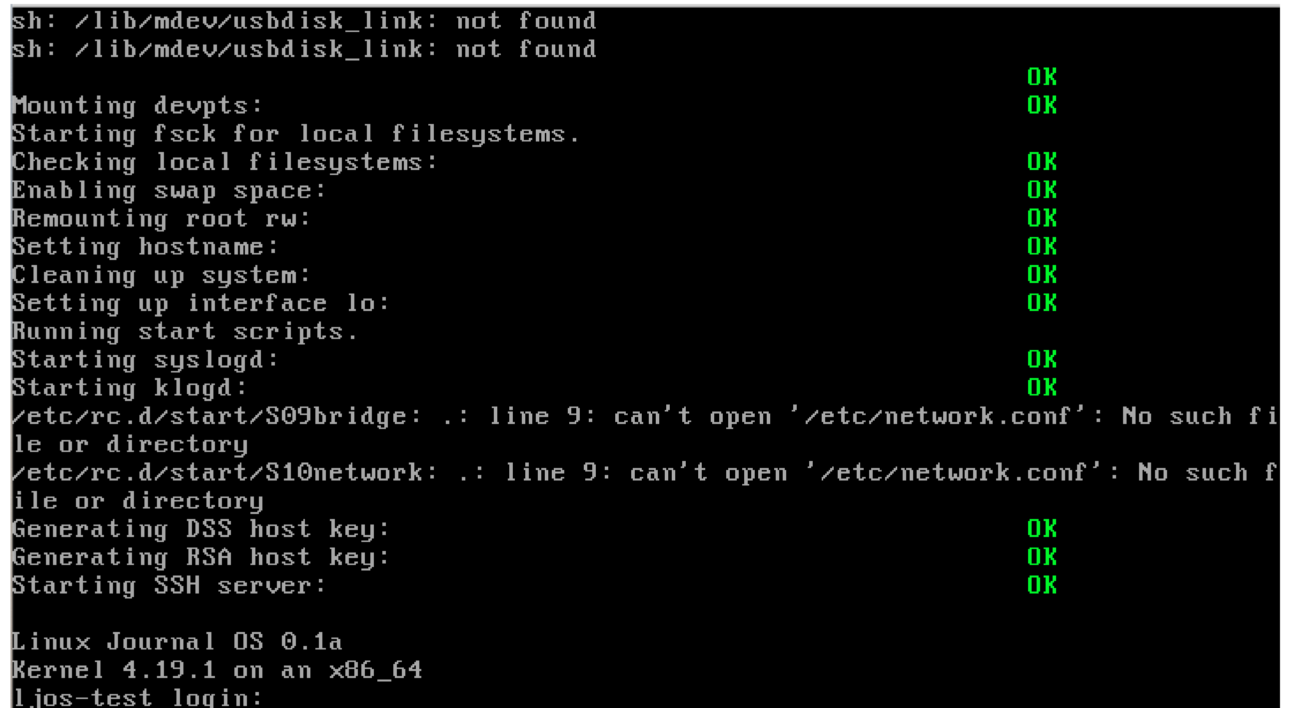


Figure 1. Errors generated during the init process of a system boot.

```
-sd[a-z].*        root:disk 0660 */lib/mdev/usbdisk_link
-hd[a-z][0-9]*    root:disk 0660 */lib/mdev/ide_links
+sd[a-z].*        root:disk 0660
+hd[a-z][0-9]*    root:disk 0660

 tty              root:tty 0666
 tty[0-9]         root:root 0600
```

Now, let's address the networking-related errors. Create the ${LJOS}/etc/network/interfaces file:

```
$ cat > ${LJOS}/etc/network/interfaces << "EOF"
> auto eth0
> iface eth0 inet dhcp
> EOF
```

Now create the ${LJOS}/etc/network.conf file with the following contents:

```
# /etc/network.conf
# Global Networking Configuration
# interface configuration is in /etc/network.d/

INTERFACE="eth0"

# set to yes to enable networking
NETWORKING=yes

# set to yes to set default route to gateway
USE_GATEWAY=no

# set to gateway IP address
GATEWAY=10.0.2.2
```

Finally, create the **udhcpc** script. **udhcpc** is a small DHCP client primarily written for minimal or embedded Linux systems. It was (or should have been) built with your BusyBox installation if you followed the steps in Part I of this series. Create the following directories:

```
$ mkdir -pv ${LJOS}/etc/network/if-{post-{up,down},
↪pre-{up,down},up,down}.d
$ mkdir -pv ${LJOS}/usr/share/udhcpc
```

Now, create the ${LJOS}/usr/share/udhcpc/default.script file with the following contents:

```
#!/bin/sh
# udhcpc Interface Configuration
# Based on http://lists.debian.org/debian-boot/2002/11/
↪msg00500.html
# udhcpc script edited by Tim Riker <Tim@Rikers.org>

[ -z "$1" ] && echo "Error: should be called from udhcpc"
 ↪&& exit 1

RESOLV_CONF="/etc/resolv.conf"
[ -n "$broadcast" ] && BROADCAST="broadcast $broadcast"
[ -n "$subnet" ] && NETMASK="netmask $subnet"

case "$1" in
    deconfig)
            /sbin/ifconfig $interface 0.0.0.0
            ;;

    renew|bound)
            /sbin/ifconfig $interface $ip $BROADCAST $NETMASK
```

```
            if [ -n "$router" ] ; then
                    while route del default gw 0.0.0.0 dev
                     ↪$interface ; do
                            true
                    done

                    for i in $router ; do
                            route add default gw $i dev
                             ↪$interface
                    done
            fi

            echo -n > $RESOLV_CONF
            [ -n "$domain" ] && echo search $domain >>
             ↪$RESOLV_CONF
            for i in $dns ; do
                    echo nameserver $i >> $RESOLV_CONF
            done
            ;;
esac

exit 0
```

Change the file's permission to enable the execution bit for all users:

```
$ chmod +x ${LJOS}/usr/share/udhcpc/default.script
```

The next time you boot up the target image (after re-preparing it), those boot errors will have disappeared.

One last thing I want to address is the root user's default shell. In my instructions from Part I, I had you set the shell to `ash`. For some odd reason, this will give you issues when attempting to `ssh` in to the distribution (via Dropbear). To avoid this, modify

```
kernel_total_size: 0x0000000001a2c000
trampoline_32bit: 0x000000000009d000
booted via startup_32()
Physical KASLR using RDRAND RDTSC...
Virtual KASLR using RDRAND RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.
Starting mdev:                                                  OK
Mounting devpts:                                                OK
Starting fsck for local filesystems.
Checking local filesystems:                                     OK
Enabling swap space:                                            OK
Remounting root rw:                                             OK
Setting hostname:                                               OK
Cleaning up system:                                             OK
Setting up interface lo:                                        OK
Running start scripts.
Starting syslogd:                                               OK
Starting klogd:                                                 OK
Starting SSH server:                                            OK

Linux Journal OS 0.1a
Kernel 4.19.1 on an x86_64
ljos-test login: _
```

Figure 2. A Cleaned-Up System Boot

the entry in the ${LJOS}/etc/passwd file so that it reads:

```
root::0:0:root:/root:/bin/sh
```

Notice the substitution of `ash` with `sh`. Ultimately, it's the same shell, as `sh` is a softlink to `ash`.

Re-Configuring the Environment

The cross-compilation build directory and the headers from the previous article should not have been deleted. Export the following variables (which you probably can throw into a script file):

```
set +h
umask 022
```

```
export LJOS=~/lj-os
export LC_ALL=POSIX
export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
unset CFLAGS
unset CXXFLAGS
export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
export LJOS_TARGET=x86_64-unknown-linux-gnu
export LJOS_CPU=k8
export LJOS_ARCH=$(echo ${LJOS_TARGET} | sed -e 's/-.*//'
 ↪-e 's/i.86/i386/')
export LJOS_ENDIAN=little
export CC="${LJOS_TARGET}-gcc"
export CXX="${LJOS_TARGET}-g++"
export CPP="${LJOS_TARGET}-gcc -E"
export AR="${LJOS_TARGET}-ar"
export AS="${LJOS_TARGET}-as"
export LD="${LJOS_TARGET}-ld"
export RANLIB="${LJOS_TARGET}-ranlib"
export READELF="${LJOS_TARGET}-readelf"
export STRIP="${LJOS_TARGET}-strip"
```

Dropbear

Dropbear is a lightweight SSH server and client. It's especially useful in minimal or embedded Linux distributions, and that's why you'll be installing it here. But before doing so, change into the CLFS bootscripts directory (clfs-embedded-bootscripts-1.0-pre5) from the previous part and install the customized init scripts:

```
$ make DESTDIR=${LJOS}/ install-dropbear
```

Now that you've installed the init scripts for Dropbear, install the SSH server and client package. Change into the package directory, and run the following configure command:

```
CC="${CC} -Os" ./configure --prefix=/usr --host=${LJOS_TARGET}
```

Compile the package:

```
$ make MULTI=1 PROGRAMS="dropbear dbclient dropbearkey
 ↪dropbearconvert scp"
```

Install the package:

```
$ make MULTI=1 PROGRAMS="dropbear dbclient dropbearkey
 ↪dropbearconvert scp" DESTDIR=${LJOS}/ install
```

Make sure the following directories are created:

```
$ mkdir -pv ${LJOS}/{etc,usr/sbin}
$ install -dv ${LJOS}/etc/dropbear
```

And, softlink the following binary:

```
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/sbin/dropbear
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/bin/dbclient
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/bin/dropbearkey
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/bin/dropbearconvert
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/bin/scp
ln -svf /usr/bin/dropbearmulti ${LJOS}/usr/bin/ssh
```

BusyBox (Revisited)

Later in this tutorial, I take a look at the HTTP dæmon included in the BusyBox package. If you haven't already, customize the package's config file to make sure that `httpd` is selected and built:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-" menuconfig
```

```
BusyBox 1.28.3 Configuration
                              Networking Utilities
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> includes, <N>
  excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]
  excluded  <M> module  < > module capable

                          ^(-)
                          [*] hostname (5.6 kb)
                          [*] dnsdomainname (3.6 kb)
                          [*] httpd (32 kb)
                          [*]   Support 'Ranges:' header
                          [*]   Enable -u <user> option
                          [*]   Enable HTTP authentication
                          [*]     Support MD5-encrypted passwords in HTTP authentication
                          [*]   Support Common Gateway Interface (CGI)
                          [*]     Support running scripts through an interpreter
                          [*]     Set REMOTE_PORT environment variable for CGI
                          [*]   Enable -e option (useful for CGIs written as shell scripts)
                          [*]   Support custom error pages
                          [*]   Support reverse proxy
                          [*]   Support GZIP content encoding
                          [*] ifconfig (12 kb)
                          [*]   Enable status reporting output (+7k)
                          [*]   Enable slip-specific options "keepalive" and "outfill"
                          [*]   Enable options "mem_start", "io_addr", and "irq"
                          [*]   Enable option "hw" (ether only)
                          [*]   Set the broadcast automatically
                          [*] ifenslave (13 kb)
```

Figure 3. The Busybox Configuration Menu

Compile and install the package:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-"
$ make CROSS_COMPILE="${LJOS_TARGET}-" \
CONFIG_PREFIX="${LJOS}" install
```

Iana-Etc

The Iana-Etc package provides your distribution with the data for the various network services and protocols as it relates to the files /etc/services and /etc/protocols. The package itself most likely will come with outdated data and IANA (Internet Assigned Numbers Authority), which is why you'll need to apply a patch written by Andrew Bradford to adjust the download location for the data update.

Change into the package directory and apply the patch:

```
$ patch -Np1 -i ../iana-etc-2.30-update-2.patch
```

Update the package's data:

```
$ make get
```

Convert the raw data and IANA into their proper formats:

```
$ make STRIP=yes
```

Install the newly created /etc/services and /etc/protocols files:

```
make DESTDIR=${LJOS} install
```

Netplug

The Netplug dæmon detects the insertion and removal of network cables and will react by bringing up or taking down the respective network interface. Similar to the Iana-Etc package, the same Andrew Bradford wrote a patch to address some issues with Netplug.

Change into the package directory and apply the patch:

```
$ patch -Np1 -i ../netplug-1.2.9.2-fixes-1.patch
```

Compile and install the package:

```
$ make && make DESTDIR=${LJOS}/ install
```

Sysstat

This is a simple one, and although you don't necessarily need this package, let's install it anyway, because it provides a nice example of how other packages are to be installed (should you choose to install more on your own). Sysstat provides a collection of monitoring utilities, which include sar, sadf, mpstat, iostat, tapestat, pidstat, cifsiostat and sa tools.

Change into the package directory and configure/compile/install the package:

```
$ ./configure --prefix=/usr --disable-documentation
$ make
$ make DESTDIR=${LJOS}/ install
```

Installing the Target Image (Again)

You'll need to create a staging area to remove unnecessary files and strip your binaries of any and all debugging symbols, but in order to do so, you'll need to copy your entire target build environment to a new location:

```
$ cp -rf ${LJOS}/ ${LJOS}-copy
```

Remove the cross-compiler toolchain and source/header files from the copy:

```
$ rm -rfv ${LJOS}-copy/cross-tools
$ rm -rfv ${LJOS}-copy/usr/src/*
```

Generate a list of all static libraries and remove them:

```
$ FILES="$(ls ${LJOS}-copy/usr/lib64/*.a)"
$ for file in $FILES; do
> rm -f $file
> done
```

Strip all debugging symbols from every binary:

```
$ find ${LJOS}-copy/{,usr/}{bin,lib,sbin} -type f -exec
↪sudo strip --strip-debug '{}' ';'
$ find ${LJOS}-copy/{,usr/}lib64 -type f -exec sudo
↪strip --strip-debug '{}' ';'
```

Change ownership of every file to root:

```
$ sudo chown -R root:root ${LJOS}-copy
```

And change the group and permissions of the following three files:

```
$ sudo chgrp 13 ${LJOS}-copy/var/run/utmp
 ↪${LJOS}-copy/var/log/lastlog
$ sudo chmod 4755 ${LJOS}-copy/bin/busybox
```

Create the following character device nodes:

```
$ sudo mknod -m 0666 ${LJOS}-copy/dev/null c 1 3
$ sudo mknod -m 0600 ${LJOS}-copy/dev/console c 5 1
```

You'll need to change into the directory of your copy and compress everything into a tarball:

```
cd ${LJOS}-copy/
sudo tar cfJ ../ljos-build-10Nov2018.tar.xz *
```

Now that you have your entire distribution archived into a single file, you'll need to move your attention to the disk volume on which it will be installed. For the rest of this tutorial, you'll need a free disk drive, and it will need to enumerate as a traditional block device (in my case, it's /dev/sdd):

```
$ cat /proc/partitions |grep sdd
   8       48     256000 sdd
```

That block device needs to be partitioned. A single partition should suffice, and you can use any one of a number of partition utilities, including `fdisk` or `parted`. Once that partition is created and detected by the host system, format the partition with an Ext4 filesystem, mount it to a staging area and change into

that directory:

```
$ sudo mkfs.ext4 /dev/sdd1
$ sudo mkdir tmp
$ sudo mount /dev/sdd1 tmp/
$ cd tmp/
```

Uncompress the operating system tarball of the entire target operating system into the root of the staging directory:

```
$ sudo tar xJf ../ljos-build-10Nov2018.tar.xz
```

Now run `grub-install` to install all the necessary modules and boot records to the volume:

```
$ sudo grub-install --root-directory=/mnt/tmp/ /dev/sdd
```

The `--root-directory` parameter defines the absolute path of the staging directory, and the last parameter is the block device without the partition's label.

Once complete, install the HDD to the physical or virtual machine, and power it up (as the primary disk drive). Within one second, you'll be at the operating system's login prompt.

Note: if you're planning to load this into a virtual machine, it'll make your life much easier if the network interface to the VM is bridged to the local Ethernet interface of your host machine.

As was the case with Part I, you never set a root password. Log in as root, and you'll immediately fall into a shell without needing to input a password. You can change this behavior by using BusyBox's `passwd` command, which should have been built in to this image. Before proceeding, change your root password.

To test the SSH dæmon, you'll need to assign an IP address to your Ethernet port. If you type `ip addr show` at the command line, you'll see that one does not exist for `eth0`. To address that, run:

```
$ udhcpc
```

The above command will work only if the `udhcpc` scripts from earlier were created and saved to the target area of your distribution. If successful, re-running `ip addr show` will show an IP address for `eth0`. In my case, the address is 192.168.1.90.

On a separate machine, log in to your LJOS distribution via SSH:

```
$ ssh root@192.168.1.90
The authenticity of host '192.168.1.90 (192.168.1.90)'
 ↪can't be established.
RSA key fingerprint is SHA256:Jp64l+7ECw2Xm5JjTXCNtEvrh
↪YRZiQzgJpBK5ljNfwk.
Are you sure you want to continue connecting (yes/no)? Yes
root@192.168.1.90's password:
~ #
```

Voilà! You're officially remotely connected.

There is so much more you can do here. Remember earlier, when I requested that you double-check that BusyBox is building its lightweight HTTP dæmon? Let's take a look at that.

Create a home directory for the dæmon:

```
# mkdir /var/www
```

And using BusyBox's lightweight `vi` program, create the /var/www/index.html file and

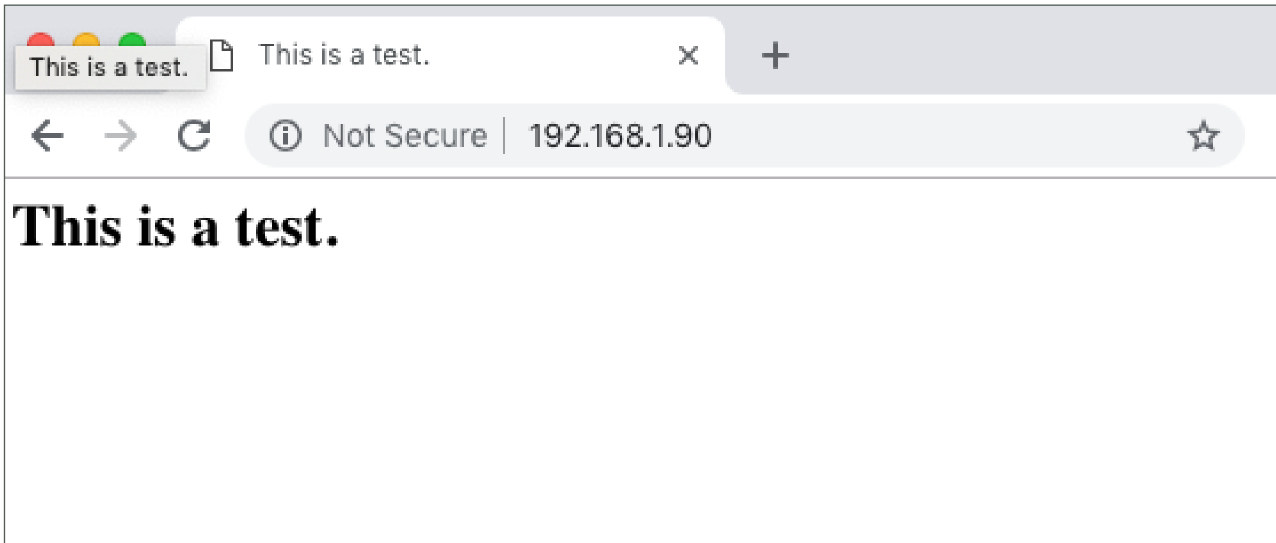


Figure 4. Accessing the Web Server Hosted from Your Custom Distribution

make sure it contains the following:

```
<html>
<head><title>This is a test.</title></head>
<body><h1>This is a test.</h1></body>
</html>
```

Save and exit. Then manually bring up the HTTP dæmon with the argument defining its home directory:

```
# httpd -h /var/www
```

Verify that the service is running:

```
# ps aux|grep http
 1177 root      0:00 httpd -h /var/www
```

On a separate machine and using your web browser, connect to the IP address of your

Linux distribution (the same address you SSH'd to). A crude HTML web page hosted by your distribution will appear.

Summary

This article builds on the exercise from my previous article and added more to the minimal and custom Linux distribution. It doesn't need to end here though. Find a purpose for it, and using the examples highlighted here, build more packages into it. ■

Petros Koutoupis, *LJ* Editor at Large, is currently a senior platform architect at IBM for its Cloud Object Storage division (formerly Cleversafe). 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 Linux Distribution from Source" by Petros Koutoupis, *Linux Journal*, June 2018
- iana-etc-2.30.patch, written by Andrew Bradford
- netplug-1.2.9.2-fixes-1.patch, written by Andrew Bradford

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

Figure 1. elementary 5 "Juno"

elementary 5 "Juno"

A review of the elementary distribution and an interview with its founders.

By Bryan Lunduke

In the spring of 2014 (nearly five years ago), I was preparing a regular presentation I give most years, where I look at the bad side (and the good side) of the greater Linux world. As I had done in years prior, I was creating a graph showing the market share of various Linux distributions changing over time.

But, this year, something was different.

In the span of less than two years, a tiny little Linux distro came out of nowhere to become one of the most watched and talked about systems available. In the blink of an eye, it went from nothing to passing several granddaddies of Linux flavors that had

been around for decades.

This was elementary. Needless to say, it caught my attention.

In the years that followed, I've interviewed elementary's founders on a few occasions—for articles, videos or podcasts—and consistently found their vision, dedication and attitudes rather intriguing.

Then in 2016, I was at a Linux conference—SCaLE (the Southern California Linux Expo). One bright, sunshiny morning, I found myself heading from my hotel room down to the conference floor. On my way, I got it in my head that I really could use some French toast. I had a hankering—a serious one. And when Lunduke gets a hankering, no force in the cosmos can stop him (he says, switching to talking about himself in the third person seemingly at random).

Somehow or another, I ended up convincing the elementary crew (four of them, also at SCaLE, with a booth to promote their system) to join me on my French toast quest.

After searching the streets of downtown Pasadena, we found ourselves in a small, but packed, diner—solving French Toast Crisis 2016—and allowing us to chat and get to know each other, in person, a bit better.

These were...kids—in their mid-20s, practically wee babies.

But, I tell you, they impressed me. Their vision for what elementary was—and what it could be—was clear. Their passion was contagious. It was hard to sit with them, in that cramped little diner, and not feel excited and optimistic for what the future held.

And, what's more, they were simply nice people. They oozed goodness and kindness. Their spirit had not yet been crushed by a string of IT managers that make soul-crushing a hobby.

They were the future of desktop Linux (or at least a rather big part of it). This was evident, even back then. And, that wasn't just the French toast talking.

When the latest release came out (elementary 5, code-name "Juno"), I got ahold of the two co-founders of elementary: Daniel Fore and Cassidy Blaede, both of whom work full time on the project as their day jobs. That's right. This is a free and open-source system, started as a passion project, that is now a small company with full-time employees working on it. It's always nice to see that sort of success in the Linux world.

First, some thoughts on elementary 5, then let's talk to the founders.

Review: elementary 5 "Juno"

I've spent some time with every release of elementary to date, and I've had some pretty glowing words for each. Yet, perhaps oddly, I've never stuck with elementary as my primary desktop system.

After a week or two with elementary I would, invariably, find my way back to the warm embrace of the likes of Debian or openSUSE—mostly, I think, due to familiarity. No matter how much I enjoyed my time with elementary, I just wasn't prepared to commit fully.

That time, I believe, has arrived.

AppCenter

One of the biggest challenges in the Linux (and greater Free and Open Source) world is how to make a good living developing software, when you're just giving the code away for free.

Some companies have pulled off this seemingly magical feat, often through paid support contracts or add-on services. Unfortunately, this tends to work really well only when your market is business-oriented, such as the enterprise offerings of Red Hat, SUSE or Canonical. Consumer-oriented applications (and games) need a

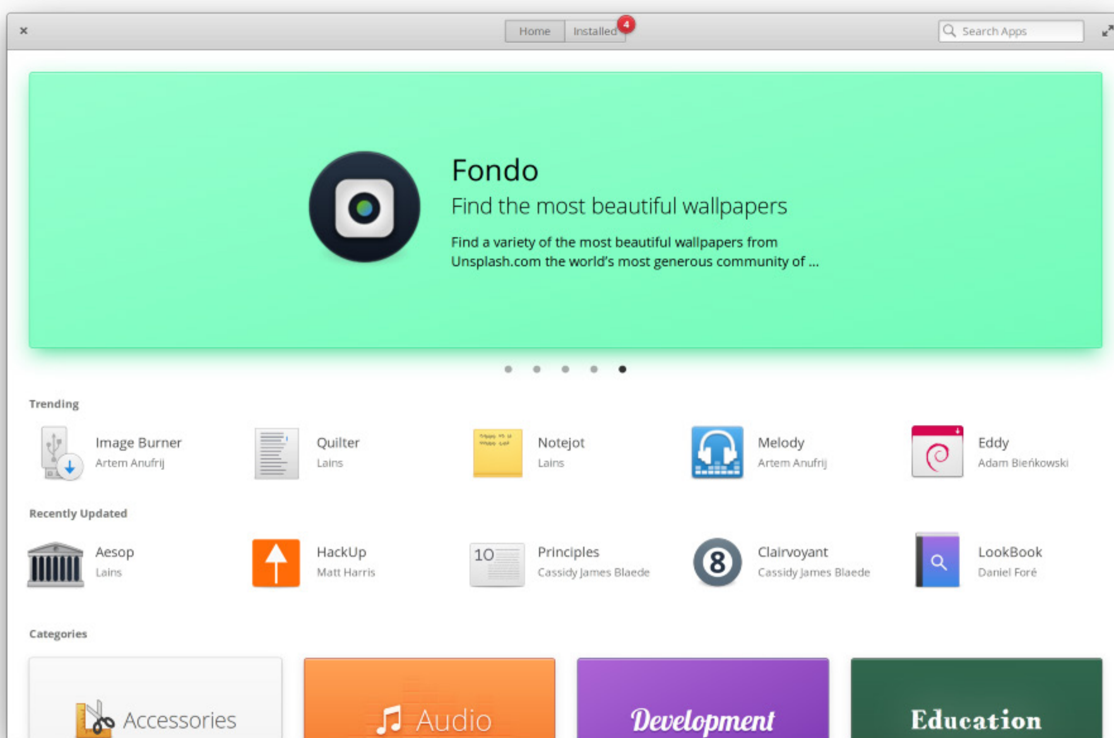


Figure 2. AppCenter

different approach.

For that, you need a place to buy the software—a place that regular people (not businesses) feel comfortable using, be it a physical brick-and-mortar store or a virtual "app store".

Through the years, there have been several attempts at building app stores for Linux. The Linspire/Lindows Click N Run store and the Ubuntu Software Center spring to mind. Both offered the ability to buy and sell Linux software, and both failed to achieve much success before being shelved entirely. Only Valve's Steam store has endured, but since Steam focuses on games (and is closed-source), there has been a major void in the market for a solution that caters to non-game software and is something that open-source and free software proponents could feel comfortable using.

The elementary team thinks they have the answer in their AppCenter.

AppCenter is, as the name suggests, a pretty traditional "app store" in most respects. There are applications organized into categories that you can search through, featured applications and update functionality—all pretty run-of-the-mill stuff.

What truly makes AppCenter exciting are two key points:

1. Every application is open source, making this a viable solution for those of us who try to avoid closed-source software.

2. Applications are "pay what you want".

That "pay what you want" bit is a pretty big deal, and it goes a long way toward making the AppCenter viable and approachable for folks used to all of their software being

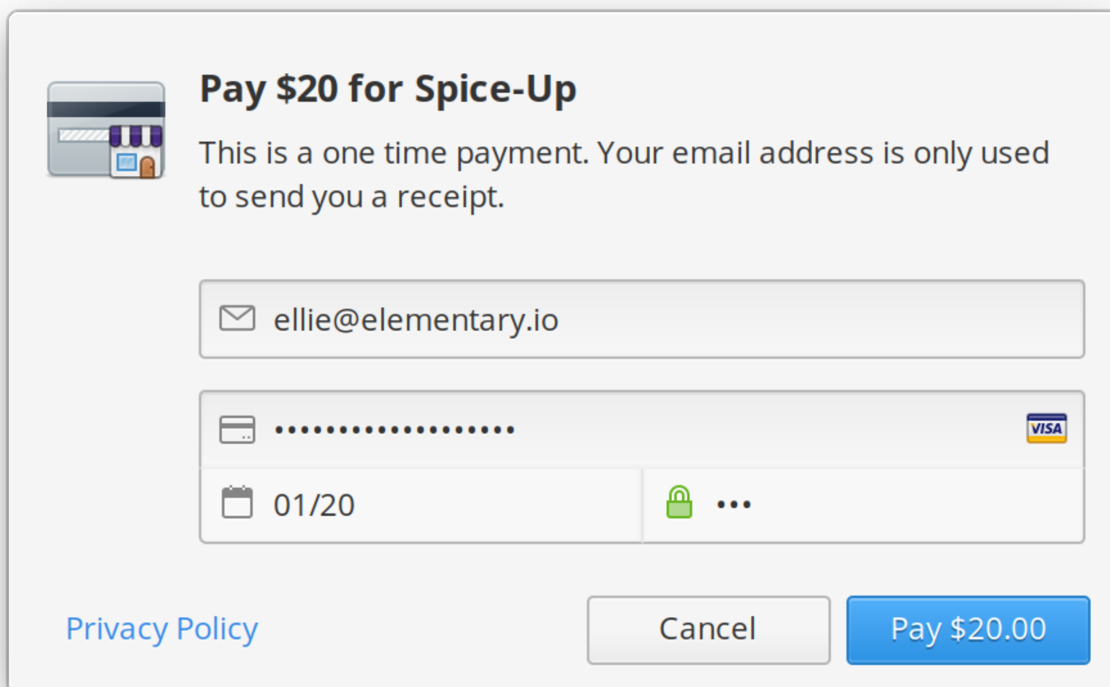


Figure 3. AppCenter, Pay What You Want

"free" (as in cost) as well as those who simply cannot afford to pay for software for whatever reason.

It also means that any application within AppCenter is, effectively, "try before you buy", as you can elect to pay zero dollars ($0.00) for something, and then, if you like it, come back later and pay the developer any price you feel the software is worth.

These two key items have been missing from every previous application store for desktop Linux. And, I believe, this is an excellent strategy from the elementary team. Given time and adequate user numbers, this could grow to become a viable revenue stream for independent Linux software developers building consumer-oriented, open-source applications and games.

And that's a very good thing.

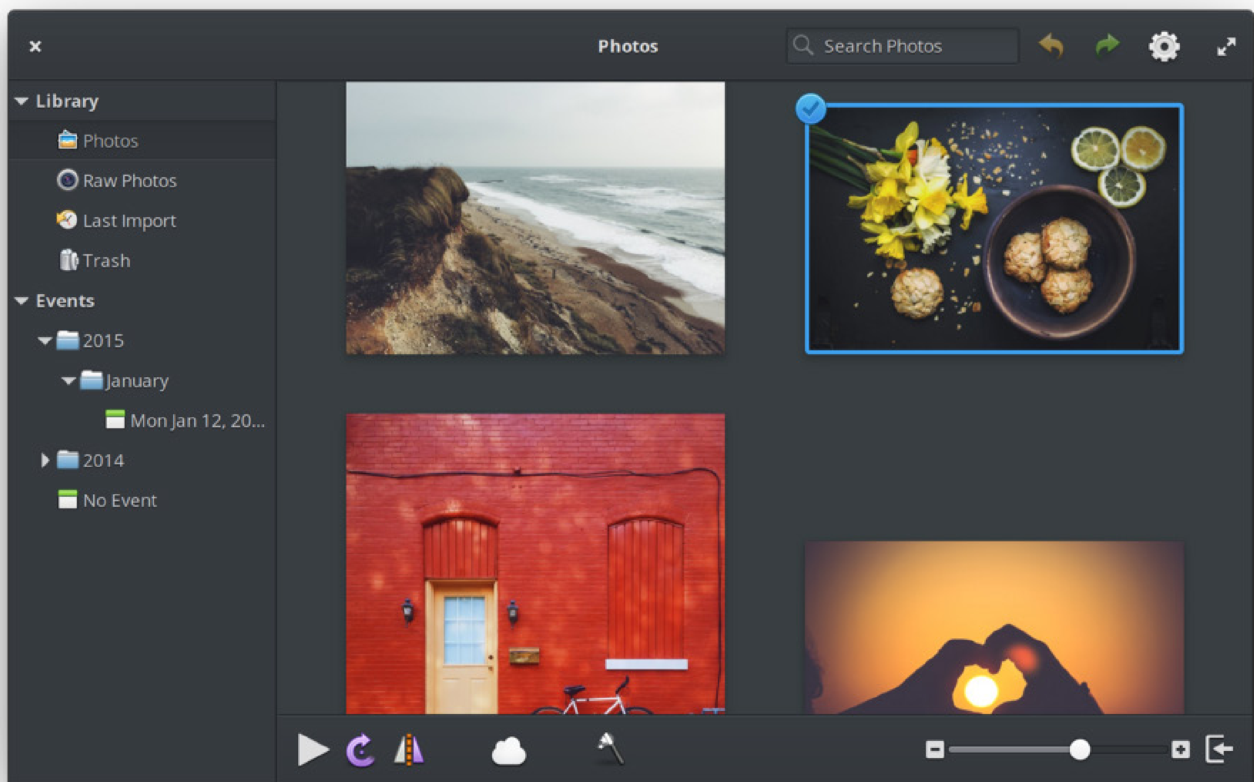


Figure 4. Photos

Applications Named What They Do

Within elementary, a large majority of the built-in software is named after exactly what the software does. Although this approach may lack originality, it makes up for it in ease of discoverability. People new to the platform will be able to find the exact piece of software needed for a task quickly. The file browser? It's called "Files". "Code" is the code editor. "Music" is the music player. "Photos" is the photo organizer.

The applications themselves are laser-focused on doing what they say they do. There tends to be very little "feature bloat" with these tools; the "Camera" application takes pictures and video—and that's it.

This modularity really speaks to my UNIX-loving sensibilities of "do one thing and do it well".

There are some applications included that don't strictly adhere to this naming scheme (such as Epiphany, the included web browser), but the exceptions are few.

Picture-in-Picture

I'm typically a pretty "traditional" person when it comes to window management on my desktop. I tend not to use features that "dock" application windows to a side of the screen or the like. But there's one new feature of elementary that I find rather fantastic. It's called "Picture-in-Picture" mode, and it works exactly like you'd expect it.

Press a hot-key, and select any window you like. That Window now becomes a small, floating, live version of itself—one that you can move and resize however you like and that stays above all other applications, on every virtual desktop (until you close it).

The obvious use for this is if you are watching a video. You can keep it playing in the corner while working on—whatever else it is you need to work on. Although I've also found it to be fun to use for things like system monitors (`htop` running in a terminal, put into Picture-in-Picture mode is oddly satisfying) and chat applications.

Configurable Shortcuts

Speaking of hot-keys (elementary calls them Shortcuts), elementary 5 adds a few

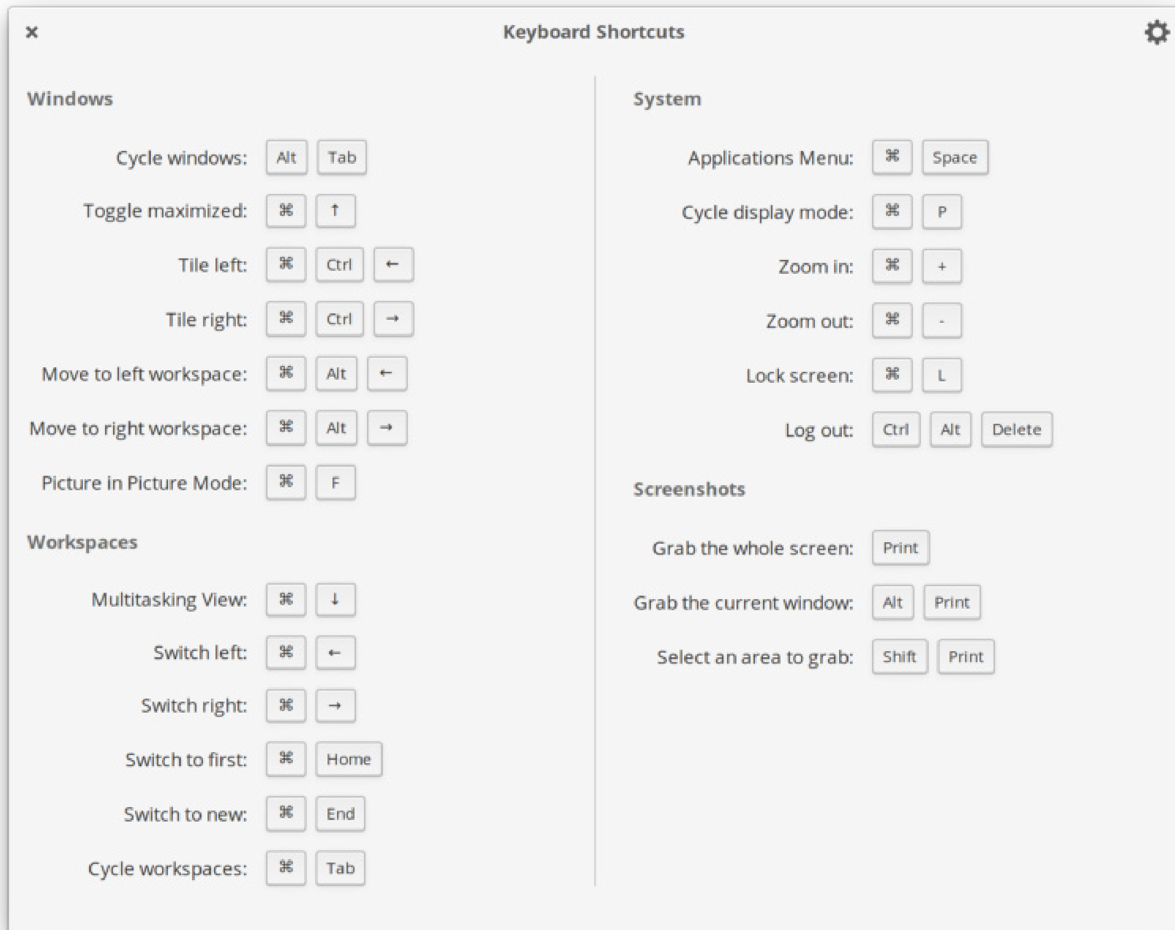


Figure 5. Shortcuts Overlay

goodies in that department.

To start with, you can tap the super key (such as the "Command" key on a Mac keyboard, or the "Windows" key) to bring up an overlay showing you the system-wide shortcuts. Handy.

Extra nice: at the top of that Shortcut Overlay is a little gear icon. Tap it and up pops the settings panel that allows you to change every Shortcut for the entire system (in case you have a bit of muscle memory from a different operating system).

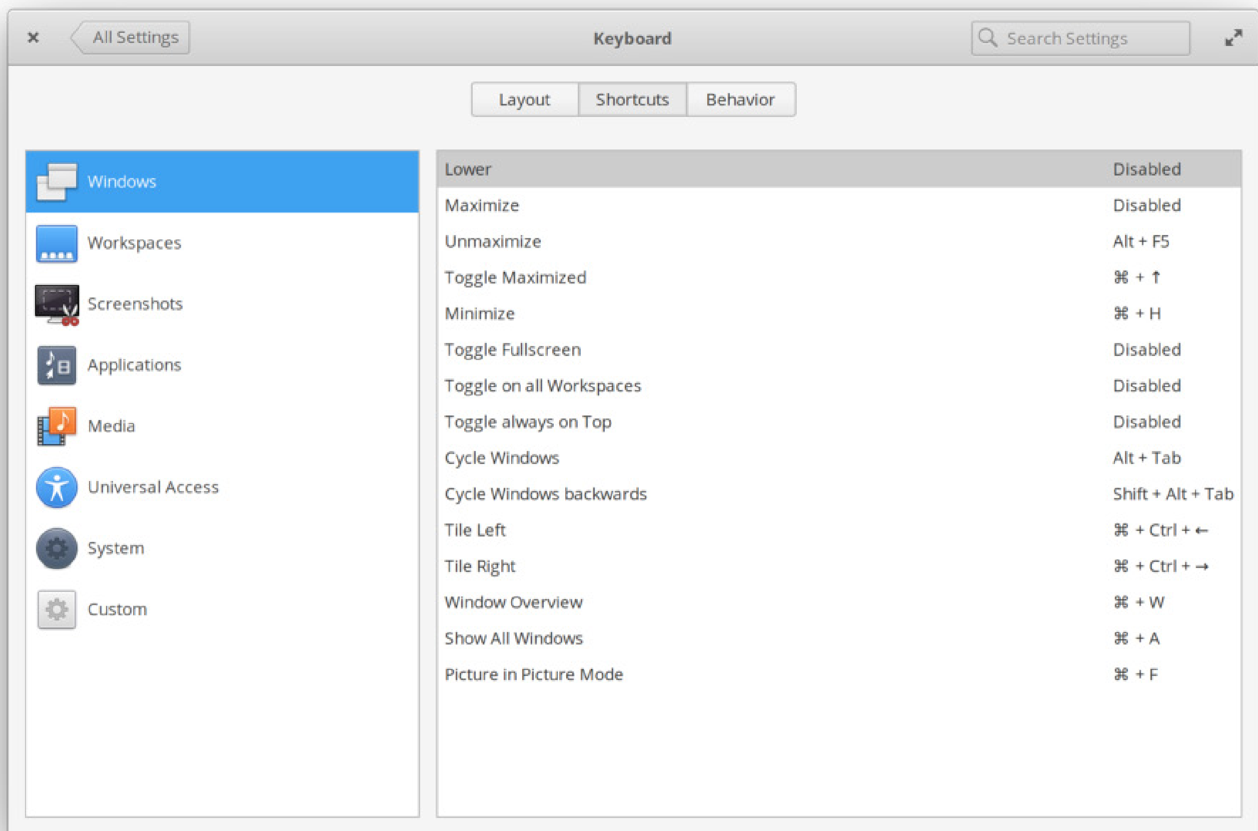


Figure 6. Shortcuts

It may seem like a little thing, but that's a level of customization that really allows people to make a system their own—to make it home.

Hardware Support

elementary makes use of the kernel and hardware support offered by Ubuntu—meaning if you have a piece of hardware that works in Ubuntu, it'll work in elementary. And, nowadays, that means pretty much everything.

I had no issues, with any component, in my testing. NVIDIA graphics cards, HDMI capture devices, wireless chipsets—everything worked right out of the box with absolutely no additional packages, tweaking or compiling required.

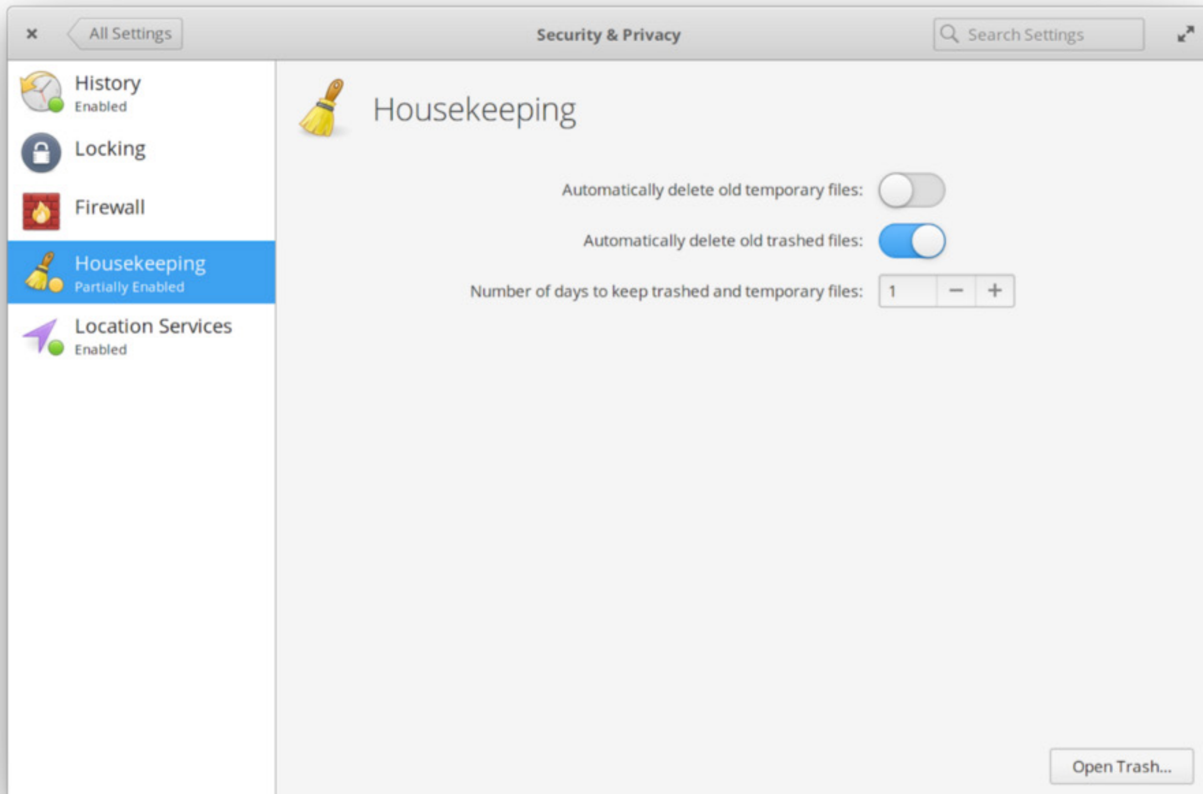


Figure 7. Housekeeping

Housekeeping

One of the cool little features of this release is a new (optional) feature called Housekeeping. It does one thing and one thing only: it deletes temporary files (or files in the trash) after a configurable number of days.

That's it, and I love that. On systems running low on storage, this is handy. But more important, it removes things you don't want around—an obvious need for keeping a system secure.

The Not-Visible Stuff

elementary has a lot of unique pieces and parts—not least of which is the desktop environment itself, Pantheon, developed specifically for the system.

But, because this is the world of Free Software, people on non-elementary

distributions want to be able to enjoy the elementary experience. In the past, getting Pantheon up and running on other systems (such as Fedora or stock Debian) was more than a small headache.

That all appears to be changing. From the elementary 5 release notes:

> ...with the Juno development cycle we've adopted more cross-desktop standards and improved the cross-distro support for several components. This came with a lot of help from Fedora maintainers and developers. The result is more reusable code for other desktops and users of other distros like Fedora, Arch, openSUSE, etc.

Working with other distributions—I like that. This is how it should be in the Open Source world.

Overall Impressions

elementary's performance is phenomenal. Interacting with the system, even on lower-end hardware, is snappy and responsive. Considering the emphasis on design and visuals, you'd almost expect the system to become a bit more sluggish (such as what people experience with Apple's Mac OS X). But there simply aren't slow-downs here. It's like you're running a lightweight system, but with all the bells and whistles of the heavier, more resource-intensive systems—an impressive feat.

Stability, likewise, has not been a concern. I've been running the release candidate version on both my laptop and my primary desktop for a week. No crashes. No hangs. Flawless.

The attention to detail throughout the system—from the application designs, to the smallest user interface items—is nothing short of impressive. Every little touch adds up to give the entire experience a polished feel.

As I wrapped up my time reviewing this release, I was left with a question I ask myself after reviewing every operating system:

Will I keep using this? Will this system replace what I've been running?

The answer is, unequivocally, yes. elementary is, in my opinion, shaping up to be one of the brightest stars in the desktop Linux sky, and I see no reason to stop using it any time soon.

Chat with elementary's Founders

After using elementary 5, I had a hodge-podge of questions for the founders (Daniel Fore and Cassidy Blaede).

Bryan Lunduke: What's the current best estimate for user base size?

Daniel Fore: Best guess is somewhere in the hundreds of thousands, likely at least 200k. It's hard to know because we don't have any sort of telemetry in the OS, and we don't generate any kind of unique fingerprint for users. The best guesses we can make is how many times packages were downloaded, which might not necessarily be unique

Daniel Fore

Cassidy Blaede

downloads and don't reflect users who haven't run updates, etc. So it could be many more or many less.

BL: How many employees are you at for the company itself, elementary, inc.?

Cassidy Blaede: Three full-time, one part-time [Daniel went full time in...] April, 2015.

BL: Have to ask—how old are you two?

CB: Personally, I'm ageless. I was here before you were born and will be here long after you're gone.

I kid, I'm 26.

DF: I'm 29.

BL: Do you remember the moment you decided elementary would be an operating system? That you needed to make it?

DF: Not really. I think there was always kind of a vision of having a complete package easily installable all together. There had been mockups of a desktop and things for quite a while. I remember the first time using Linux though and knowing that this thing was the future, and that this was the technology that was going to enable me to build the desktop I wanted to see. It was like going from drawing in black and white to color.

BL: What was that moment? And what Linux distribution did you use?

DF: It was the Kororaa LiveCD and XGL demo with Compiz. Coming from Windows XP, it was just amazing to see what you could do and that all the pieces were there, and you could build whatever you wanted.

That had to be in like 2006 or 2007.

BL: Does that mean we'll get wobbly windows and 3D virtual desktop cubes in elementary?

DF: Haha probably not. I think a lot of those flashy effects were interesting for a time when the technology was new, but motion design has matured a lot since then, and now when we use animations, we're using them to convey something meaningful or to provide hints or affordances about the way the UI can be interacted with. Now, users expect animations to be fast and not interrupt their workflow.

BL: Cassidy, talk some sense into Daniel. Wobbly windows for life (tm).

CB: Ha, my first Linux experience was actually a Knoppix live CD, but the first long-term version I used was Ubuntu. I too always would go in and enable the fancy compositor and turn on wobbly windows and exploding closing windows and all that jazz. And in the early days of elementary OS, Compiz actually let us really fine-grain control the motion design of windows and workspaces without having to write code or make our own window manager. So that was a huge boon to getting started. But these days, we have accelerated, composited windowing libraries by default, plus we actually develop the window manager itself. So we can decide exactly how to animate things so that it's all self-consistent.

Wobbly windows were a great tech demo, but I think the performance and, uh, "taste" trade-offs might not be quite up our alley.

BL: Okay. Now an open-ended question: where from here?

DF: To the cloud!

CB: Juno is really about nailing a lot of aspects about elementary OS: refining it, making it more productive and making it even better for developers. But like Dan alluded to, we're also laying the groundwork for much better online accounts integration so users can access their data and accounts no matter where they're stored.

(I also don't know if Dan was just joking or not, but there you go.)

DF: There's been a lot of work with upstream projects on things like Online Accounts and Evolution Data Server. We're working on a brand-new version of Mail powered by EDS. There's work on showing cloud storage devices in Files. We're working on a new installation and onboarding experience, so we'll hopefully get those accounts connected right from the beginning. So overall, I think the next big moves for the whole desktop will be making it much more connected to all the online services that our users are interested in.

CB: There's also AppCenter: we've laid a great foundation for our app ecosystem, and now it's time to really dial in that experience for users. We're always working on ways to enhance discovery of existing apps and also engaging with app developers to bring even more quality apps to the platform.

DF: Cassidy blogged recently about digital well being and how some of our next steps down that path involve content controls, so that's another thing we're really interested in is giving our users control over the kind of content that they and the ones they're responsible for see.

BL: Can you say, at this point, what the best-selling app in your AppCenter is?

DF: I'm having a tough time with trying to see if I can get stats for this, but I'm fairly confident the top grossing app is the torrent client Torrential by David Hewitt.

BL: What's the moment where you realized elementary had "made it" as an operating system/distro?

CB: So that's a fun question. I think there are still pretty frequent days when impostor syndrome picks up, and there's this nagging voice telling myself that we're nobodies and are never going to "make it". But on the flip side, there are days when I see that there are hundreds of thousands of people out there using something I've helped make, and that's pretty humbling. I don't know if there has really been an individual moment for me; it's more like a pendulum constantly swinging between those two mental states.

I do remember the first time we got an interview in a physical magazine, and I went to

the bookstore where my girlfriend worked, and it was right there on the shelf. That was pretty wild and felt like we'd made it in some sense.

DF: Yeah, I agree with Cassidy that a lot of times I feel like we haven't "made it", and there's so much to go before we can possibly try to compete with the major proprietary operating systems. But then you see a tweet about how elementary OS is being used in a school, or you interact with someone new at a conference who has heard of it or used it. I heard from my cousin recently that her new boyfriend used elementary OS, and seeing stuff like that is really encouraging and makes me feel like we've made a difference and an impact.

CB: Oh yeah, running into users outside an expected situation is always super encouraging. A friend of mine is a teacher, and she recently messaged me that one of her students was using elementary OS and showing it to her. He was super impressed when she told him she hadn't just heard of it, but she knew one of the founders!

So I guess "making it" isn't really this binary thing. We don't have to dethrone somebody to have made it, but we're making it more and more every day. ■

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.

Resources

- elementary OS
- AppCenter
- Get elementary OS
- Torrential by David Hewitt

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

A Use Case for Network Automation

Use the Python Netmiko module to automate switches, routers and firewalls from multiple vendors.

By Eric Pearce

I frequently find myself in the position of confronting "hostile" networks. By hostile, I mean that there is no existing documentation, or if it does exist, it is hopelessly out of date or being hidden deliberately. With that in mind, in this article, I describe the tools I've found useful to recover control, audit, document and automate these networks. Note that I'm not going to try to document any of the tools completely here. I mainly want to give you enough real-world examples to prove how much time and effort you could save with these tools, and I hope this article motivates you to explore the official documentation and example code.

In order to save money, I wanted to use open-source tools to gather information from all the devices on the network. I haven't found a single tool that works with all the vendors and OS versions that typically are encountered. SNMP could provide a lot the information I need, but it would have to be configured on each device manually first. In fact, the mass enablement of SNMP could be one of the first use cases for the network automation tools described in this article.

Most modern devices support REST APIs, but companies typically are saddled with lots of legacy devices that don't support anything fancier than Telnet and SSH. I settled on SSH access as the lowest common denominator, as every device must support this in order to be managed on the network.

My preferred automation language is Python, so the next problem was finding

a Python module that abstracted the SSH login process, making it easy to run commands and gather command output.

Why Netmiko?

I discovered the Paramiko SSH module quite a few years ago and used it to create real-time inventories of Linux servers at multiple companies. It enabled me to log in to hosts and gather the output of commands, such as `lspci`, `dmidecode` and `lsmod`.

The command output populated a database that engineers could use to search for specific hardware. When I then tried to use Paramiko to inventory network switches, I found that certain switch vendor and OS combinations would cause Paramiko SSH sessions to hang. I could see that the SSH login itself was successful, but the session would hang right after the login. I never was able to determine the cause, but I discovered Netmiko while researching the hanging problem. When I replaced all my Paramiko code with Netmiko code, all my session hanging problems went away, and I haven't looked back since. Netmiko also is optimized for the network device management task, while Paramiko is more of a generic SSH module.

Programmatically Dealing with the Command-Line Interface

People familiar with the "Expect" language will recognize the technique for sending a command and matching the returned CLI prompts and command output to determine whether the command was successful. In the case of most network devices, the CLI prompts change depending on whether you're in an unprivileged mode, in "enable" mode or in "config" mode.

For example, the CLI prompt typically will be the device hostname followed by specific characters.

Unprivileged mode:

```
sfo03-r7r9-sw1>
```

Privileged or “enable” mode:

```
sfo03-r7r9-sw1#
```

“Config” mode:

```
sfo03-r7r9-sw1(config)#
```

These different prompts enable you to make transitions programmatically from one mode to another and determine whether the transitions were successful.

Abstraction

Netmiko abstracts many common things you need to do when talking to switches. For example, if you run a command that produces more than one page of output, the switch CLI typically will “page” the output, waiting for input before displaying the next page. This makes it difficult to gather multipage output as single blob of text. The command to turn off paging varies depending on the switch vendor. For example, this might be `terminal length 0` for one vendor and `set cli pager off` for another. Netmiko abstracts this operation, so all you need to do is use the `disable_paging()` function, and it will run the appropriate commands for the particular device.

Dealing with a Mix of Vendors and Products

Netmiko supports a growing list of network vendor and product combinations. You can find the current list in the documentation. Netmiko doesn’t auto-detect the vendor, so you’ll need to specify that information when using the functions. Some vendors have product lines with different CLI commands. For example, Dell has two types: `dell_force10` and `dell_powerconnect`; and Cisco has several CLI versions on the different product lines, including `cisco_ios`, `cisco_nxos` and `cisco_asa`.

Obtaining Netmiko

The official Netmiko code and documentation is at https://github.com/ktbyers/netmiko, and the author has a collection of helpful articles on his home page.

If you're comfortable with developer tools, you can clone the GIT repo directly. For typical end users, installing Netmiko using `pip` should suffice:

```
# pip install netmiko
```

A Few Words of Caution

Before jumping on the network automation bandwagon, you need to sort out the following:

- Mass configuration: be aware that the slowness of traditional "box-by-box" network administration may have protected you somewhat from massive mistakes. If you manually made a change, you typically would be alerted to a problem after visiting only a few devices. With network automation tools, you can render all your network devices useless within seconds.

- Configuration backup strategy: this ideally would include a versioning feature, so you can roll back to a specific "known good" point in time. Check out the RANCID package before you spend a lot of money on this capability.

- Out-of-band network management: almost any modern switch or network device is going to have a dedicated OOB port. This physically separate network permits you to recover from configuration mistakes that potentially could cut you off from the very devices you're managing.

- A strategy for testing: for example, have a dedicated pool of representative equipment permanently set aside for testing and proof of concepts. When rolling out a change on a production network, first verify the automation on a few devices before trying to do hundreds at once.

Using Netmiko without Writing Any Code

Netmiko's author has created several standalone scripts called Netmiko Tools that you can use without writing any Python code. Consult the official documentation for details, as I offer only a few highlights here.

At the time of this writing, there are three tools: netmiko-show, netmiko-cfg and netmiko-grep.

netmiko-show

Run arbitrary "show" commands on one or more devices. By default, it will display the entire configuration, but you can supply an alternate command with the `--cmd` option. Note that "show" commands can display many details that aren't stored within the actual device configurations.

For example, you can display Spanning Tree Protocol (STP) details from multiple devices:

```
% netmiko-show --cmd "show spanning-tree detail" arista-eos |
 ↪egrep "(last change|from)"
sfo03-r1r12-sw1.txt:  Number of topology changes 2307 last
 ↪change occurred 19:14:09 ago
sfo03-r1r12-sw1.txt:          from Ethernet1/10/2
sfo03-r1r12-sw2.txt:  Number of topology changes 6637 last
 ↪change occurred 19:14:09 ago
sfo03-r1r12-sw2.txt:          from Ethernet1/53
```

This information can be very helpful when tracking down the specific switch and switch port responsible for an STP flapping issue. Typically, you would be looking for a very high count of topology changes that is rapidly increasing, with a "last change time" in seconds. The "from" field gives you the source port of the change, enabling you to narrow down the source of the problem.

The "old-school" method for finding this information would be to log in to the top-most switch, look at its STP detail, find the problem port, log in to the switch downstream of this port, look at its STP detail and repeat this process until you find the source of the problem. The Netmiko Tools allow you to perform a network-wide search for all the information you need in a single operation.

netmiko-cfg

Apply snippets of configurations to one or more devices. Specify the configuration command with the `--cmd` option or read configuration from a file using `--infile`. This could be used for mass configurations. Mass changes could include DNS servers, NTP servers, SNMP community strings or syslog servers for the entire network. For example, to configure the read-only SNMP community on all of your Arista switches:

```
$ netmiko-cfg --cmd "snmp-server community mysecret ro"
 ↪arista-eos
```

You still will need to verify that the commands you're sending are appropriate for the vendor and OS combinations of the target devices, as Netmiko will not do all of this work for you. See the "groups" mechanism below for how to apply vendor-specific configurations to only the devices from a particular vendor.

netmiko-grep

Search for a string in the configuration of multiple devices. For example, verify the current syslog destination in your Arista switches:

```
$ netmiko-grep --use-cache "logging host" arista-eos
sfo03-r2r7-sw1.txt:logging host 10.7.1.19
sfo03-r3r14-sw1.txt:logging host 10.8.6.99
sfo03-r3r16-sw1.txt:logging host 10.8.6.99
sfo03-r4r18-sw1.txt:logging host 10.7.1.19
```

All of the Netmiko tools depend on an "inventory" of devices, which is a YAML-formatted file stored in ".netmiko.yml" in the current directory or your home directory.

Each device in the inventory has the following format:

```
sfo03-r1r11-sw1:
  device_type: cisco_ios
  ip: sfo03-r1r11-sw1
```

```
  username: netadmin
  password: secretpass
  port: 22
```

Device entries can be followed by group definitions. Groups are simply a group name followed by a list of devices:

```
cisco-ios:
  - sfo03-r1r11-sw1
cisco-nxos:
  - sfo03-r1r12-sw2
  - sfo03-r3r17-sw1
arista-eos:
  - sfo03-r1r10-sw2
  - sfo03-r6r6-sw1
```

For example, you can use the group name "cisco-nxos" to run Cisco Nexus NX-OS-unique commands, such as `feature`:

```
% netmiko-cfg --cmd "feature interface-vlan" cisco-nxos
```

Note that the device type example is just one type of group. Other groups could indicate physical location ("SFO03", "RKV02"), role ("TOR", "spine", "leaf", "core"), owner ("Eng", "QA") or any other categories that make sense to you.

As I was dealing with hundreds of devices, I didn't want to create the YAML-formatted inventory file by hand. Instead, I started with a simple list of devices and the corresponding Netmiko "device_type":

```
sfo03-r1r11-sw1,cisco_ios
sfo03-r1r12-sw2,cisco_nxos
sfo03-r1r10-sw2,arista_eos
sfo03-r4r5-sw3,arista_eos
```

```
sfo03-r1r12-sw1,cisco_nxos
sfo03-r5r15-sw2,dell_force10
```

I then used standard Linux commands to create the YAML inventory file:

```
% grep -v '^#' simplelist.txt | awk -F, '{printf("%s:\n
 ↪device_type:
%s\n  ip: %s\n  username: netadmin\n  password:
 ↪secretpass\n  port:
22\n",$1,$2,$1)}' >> .netmiko.yml
```

I'm using a centralized authentication system, so the user name and password are the same for all devices. The command above yields the following YAML-formatted file:

```
sfo03-r1r11-sw1:
  device_type: cisco_ios
  ip: sfo03-r1r11-sw1
  username: netadmin
  password: secretpass
  port: 22
sfo03-r1r12-sw2:
  device_type: cisco_nxos
  ip: sfo03-r1r12-sw2
  username: netadmin
  password: secretpass
  port: 22
sfo03-r1r10-sw2:
  device_type: arista_eos
  ip: sfo03-r1r10-sw2
  username: netadmin
  password: secretpass
  port: 22
```

Once you've created this inventory, you can use the Netmiko Tools against individual devices or groups of devices.

A side effect of creating the inventory is that you now have a master list of devices on the network; you also have proven that the device names are resolvable via DNS and that you have the correct login credentials. This is actually a big step forward in some environments where I've worked.

Note that `netmiko-grep` caches the device configs locally. Once the cache has been built, you can make subsequent search operations run much faster by specifying the `--use-cache` option.

It now should be apparent that you can use Netmiko Tools to do a lot of administration and automation without writing any Python code. Again, refer to official documentation for all the options and more examples.

Start Coding with Netmiko

Now that you have a sense of what you can do with Netmiko Tools, you'll likely come up with unique scenarios that require actual coding.

For the record, I don't consider myself an advanced Python programmer at this time, so the examples here may not be optimal. I'm also limiting my examples to snippets of code rather than complete scripts. The example code is using Python 2.7.

My Approach to the Problem

I wrote a bunch of code before I became aware of the Netmiko Tools commands, and I found that I'd duplicated a lot of their functionality. My original approach was to break the problem into two separate phases. The first phase was the "scanning" of the switches and storing their configurations and command output locally. The second phase was processing and searching across the stored data.

My first script was a "scanner" that reads a list of switch hostnames and

Netmiko device types from a simple text file, logs in to each switch, runs a series of CLI commands and then stores the output of each command in text files for later processing.

Reading a List of Devices

My first task is to read a list of network devices and their Netmiko "device type" from a simple text file in the CSV format. I include the csv module, so I can use the csv.Dictreader function, which returns CSV fields as a Python dictionary. I like the CSV file format, as anyone with limited UNIX/Linux skills likely knows how to work with it, and it's a very common file type for exporting data if you have an existing database of network devices.

For example, the following is a list of switch names and device types in CSV format:

```
sfo03-r1r11-sw1,cisco_ios
sfo03-r1r12-sw2,cisco_nxos
sfo03-r1r10-sw2,arista_eos
sfo03-r4r5-sw3,arista_eos
sfo03-r1r12-sw1,cisco_nxos
sfo03-r5r15-sw2,dell_force10
```

The following Python code reads the data filename from the command line, opens the file and then iterates over each device entry, calling the `login_switch()` function that will run the actual Netmiko code:

```
import csv
import sys
import logging
def main():
# get data file from command line
   devfile = sys.argv[1]
# open file and extract the two fields
   with open(devfile,'rb') as devicesfile:
       fields = ['hostname','devtype']
```

```
       hosts = csv.DictReader(devicesfile,fieldnames=fields,
↪delimiter=',')
# iterate through list of hosts, calling "login_switch()"
# for each one
       for host in hosts:
           hostname = host['hostname']
           print "hostname = ",hostname
           devtype = host['devtype']
           login_switch(hostname,devtype)
```

The `login_switch()` function runs any number of commands and stores the output in separate text files under a directory based on the name of the device:

```
# import required module
from netmiko import ConnectHandler
# login into switch and run command
def login_switch(host,devicetype):
# required arguments to ConnectHandler
    device = {
# device_type and ip are read from data file
    'device_type': devicetype,
    'ip':host,
# device credentials are hardcoded in script for now
    'username':'admin',
    'password':'secretpass',
    }
# if successful login, run command on CLI
    try:
        net_connect = ConnectHandler(**device)
        commands = "show version"
        output = net_connect.send_command(commands)
# construct directory path based on device name
        path = '/root/login/scan/' + host + "/"
```

```
        make_dir(path)
        filename = path + "show_version"
# store output of command in file
        handle = open (filename,'w')
        handle.write(output)
        handle.close()
# if unsuccessful, print error
    except Exception as e:
        print "RAN INTO ERROR "
        print "Error: " + str(e)
```

This code opens a connection to the device, executes the `show version` command and stores the output in /root/login/scan/<devicename>/show_version.

The `show version` output is incredibly useful, as it typically contains the vendor, model, OS version, hardware details, serial number and MAC address. Here's an example from an Arista switch:

```
Arista DCS-7050QX-32S-R
Hardware version:    01.31
Serial number:       JPE16292961
System MAC address:  444c.a805.6921

Software image version: 4.17.0F
Architecture:           i386
Internal build version: 4.17.0F-3304146.4170F
Internal build ID:      21f25f02-5d69-4be5-bd02-551cf79903b1

Uptime:                 25 weeks, 4 days, 21 hours and 32
                        minutes
Total memory:           3796192 kB
Free memory:            1230424 kB
```

This information allows you to create all sorts of good stuff, such as a hardware inventory of your network and a software version report that you can use for audits and planned software updates.

My current script runs `show lldp neighbors`, `show run`, `show interface status` and records the device CLI prompt in addition to `show version`.

The above code example constitutes the bulk of what you need to get started with Netmiko. You now have a way to run arbitrary commands on any number of devices without typing anything by hand. This isn't Software-Defined Networking (SDN) by any means, but it's still a huge step forward from the "box-by-box" method of network administration.

Next, let's try the scanning script on the sample network:

```
$ python scanner.py devices.csv
hostname = sfo03-r1r15-sw1
hostname = sfo03-r3r19-sw0
hostname = sfo03-r1r16-sw2
hostname = sfo03-r3r8-sw2
RAN INTO ERROR
Error: Authentication failure: unable to connect dell_force10
 ↪sfo03-r3r8-sw2:22
Authentication failed.
hostname = sfo03-r3r10-sw2
hostname = sfo03-r3r11-sw1
hostname = sfo03-r4r14-sw2
hostname = sfo03-r4r15-sw1
```

If you have a lot of devices, you'll likely experience login failures like the one in the middle of the scan above. These could be due to multiple reasons, including the device being down, being unreachable over the network, the script having incorrect credentials and so on. Expect to make several passes to address all the problems

before you get a "clean" run on a large network.

This finishes the "scanning" portion of process, and all the data you need is now stored locally for further analysis in the "scan" directory, which contains subdirectories for each device:

```
$ ls scan/
sfo03-r1r10-sw2 sfo03-r2r14-sw2 sfo03-r3r18-sw1 sfo03-r4r8-sw2
 ↪sfo03-r6r14-sw2
sfo03-r1r11-sw1 sfo03-r2r15-sw1 sfo03-r3r18-sw2 sfo03-r4r9-sw1
 ↪sfo03-r6r15-sw1
sfo03-r1r12-sw0 sfo03-r2r16-sw1 sfo03-r3r19-sw0 sfo03-r4r9-sw2
 ↪sfo03-r6r16-sw1
sfo03-r1r12-sw1 sfo03-r2r16-sw2 sfo03-r3r19-sw1 sfo03-r5r10-sw1
 ↪sfo03-r6r16-sw2
sfo03-r1r12-sw2 sfo03-r2r2-sw1  sfo03-r3r4-sw2  sfo03-r5r10-sw2
 ↪sfo03-r6r17- sw1
```

You can see that each subdirectory contains separate files for each command output:

```
$ ls sfo03-r1r10-sw2/
show_lldp prompt show_run show_version show_int_status
```

Debugging via Logging

Netmiko normally is very quiet when it's running, so it's difficult to tell where things are breaking in the interaction with a network device. The easiest way I have found to debug problems is to use the logging module. I normally keep this disabled, but when I want to turn on debugging, I uncomment the line starting with `logging.basicConfig` line below:

```
import logging
if __name__ == "__main__":
#  logging.basicConfig(level=logging.DEBUG)
  main()
```

Then I run the script, and it produces output on the console showing the entire SSH conversation between the netmiko module and the remote device (a switch named "sfo03-r1r10-sw2" in this example):

```
DEBUG:netmiko:In disable_paging
DEBUG:netmiko:Command: terminal length 0
DEBUG:netmiko:write_channel: terminal length 0
DEBUG:netmiko:Pattern is: sfo03\-r1r10\-sw2
DEBUG:netmiko:_read_channel_expect read_data: terminal
 ↪length 0
DEBUG:netmiko:_read_channel_expect read_data: Pagination
disabled.
sfo03-r1r10-sw2#
DEBUG:netmiko:Pattern found: sfo03\-r1r10\-sw2 terminal
 ↪length 0
Pagination disabled.
sfo03-r1r10-sw2#
DEBUG:netmiko:terminal length 0
Pagination disabled.
sfo03-r1r10-sw2#
DEBUG:netmiko:Exiting disable_paging
```

In this case, the `terminal length 0` command sent by Netmiko is successful. In the following example, the command sent to change the terminal width is rejected by the switch CLI with the "Authorization denied" message:

```
DEBUG:netmiko:Entering set_terminal_width
DEBUG:netmiko:write_channel: terminal width 511
DEBUG:netmiko:Pattern is: sfo03\-r1r10\-sw2
DEBUG:netmiko:_read_channel_expect read_data: terminal
 ↪width 511
DEBUG:netmiko:_read_channel_expect read_data: % Authorization
denied for command 'terminal width 511'
```

```
sfo03-r1r10-sw2#
DEBUG:netmiko:Pattern found: sfo3\-r1r10\-sw2 terminal
 ↪width 511
% Authorization denied for command 'terminal width 511'
sfo03-r1r10-sw2#
DEBUG:netmiko:terminal width 511
% Authorization denied for command 'terminal width 511'
sfo03-r1r10-sw2#
DEBUG:netmiko:Exiting set_terminal_width
```

The logging also will show the entire SSH login and authentication sequence in detail. I had to deal with one switch that was using a depreciated SSH cypher that was disabled by default in the SSH client, causing the SSH session to fail when trying to authenticate. With logging, I could see the client rejecting the cypher being offered by the switch. I also discovered another type of switch where the Netmiko connection appeared to hang. The logging revealed that it was stuck at the `more?` prompt, as the paging was never disabled successfully after login. On this particular switch, the commands to disable paging had to be run in a privileged mode. My quick fix was adding a `disable_paging()` function after the "enable" mode was entered.

Analysis Phase

Now that you have all the data you want, you can start processing it.

A very simple example would be an "audit"-type of check, which verifies that the hostname registered in DNS matches the hostname configured in the device. If these do not match, it will cause all sorts of confusion when logging in to the device, correlating syslog messages or looking at LLDP and CPD output:

```
import os
import sys
directory = "/root/login/scan"
for filename in os.listdir(directory):
    prompt_file = directory + '/' + filename + '/prompt'
```

```
    try:
         prompt_fh = open(prompt_file,'rb')
    except IOError:
         "Can't open:", prompt_file
         sys.exit()

    with prompt_fh:
        prompt = prompt_fh.read()
        prompt = prompt.rstrip('#')
        if (filename != prompt):
            print 'switch DNS hostname %s != configured
             ↪hostname %s' %(filename, prompt)
```

This script opens the scan directory, opens each "prompt" file, derives the configured hostname by stripping off the "#" character, compares it with the subdirectory filename (which is the hostname according to DNS) and prints a message if they don't match. In the example below, the script finds one switch where the DNS switch name doesn't match the hostname configured on the switch:

```
$ python name_check.py
switch DNS hostname sfo03-r1r12-sw2 != configured hostname
 ↪SFO03-R1R10-SW1-Cisco_Core
```

It's a reality that most complex networks are built up over a period of years by multiple people with different naming conventions, work styles, skill sets and so on. I've accumulated a number of "audit"-type checks that find and correct inconsistencies that can creep into a network over time. This is the perfect use case for network automation, because you can see everything at once, as opposed going through each device, one at a time.

Performance

During the initial debugging, I had the "scanning" script log in to each switch in a serial fashion. This worked fine for a few switches, but performance became a problem

when I was scanning hundreds at a time. I used the Python multiprocessing module to fire off a bunch of "workers" that interacted with switches in parallel. This cut the processing time for the scanning portion down to a couple minutes, as the entire scan took only as long as the slowest switch took to complete. The switch scanning problem fits quite well into the multiprocessing model, because there are no events or data to coordinate between the individual workers. The Netmiko Tools also take advantage of multiprocessing and use a cache system to improve performance.

Future Directions

The most complicated script I've written so far with Netmiko logs in to every switch, gathers the LLDP neighbor info and produces a text-only topology map of the entire network. For those who are unfamiliar with LLDP, this is the Link Layer Discovery Protocol. Most modern network devices are sending LLDP multicasts out every port every 30 seconds. The LLDP data includes many details, including the switch hostname, port name, MAC address, device model, vendor, OS and so on. It allows any given device to know about all its immediate neighbors.

For example, here's a typical LLDP display on a switch. The "Neighbor" columns show you details on what is connected to each of your local ports:

```
sfo03-r1r5-sw1# show lldp neighbors
Port  Neighbor Device ID   Neighbor Port ID   TTL
Et1   sfo03-r1r3-sw1         Ethernet1          120
Et2   sfo03-r1r3-sw2         Te1/0/2            120
Et3   sfo03-r1r4-sw1         Te1/0/2            120
Et4   sfo03-r1r6-sw1         Ethernet1          120
Et5   sfo03-r1r6-sw2         Te1/0/2            120
```

By asking all the network devices for their list of LLDP neighbors, it's possible to build a map of the network. My approach was to build a list of local switch ports and their LLDP neighbors for the top-level switch, and then recursively follow each switch link down the hierarchy of switches, adding each entry to a nested dictionary. This process becomes very complex when there are redundant links and endless loops to

avoid, but I found it a great way to learn more about complex Python data structures.

The following output is from my "mapper" script. It uses indentation (from left to right) to show the hierarchy of switches, which is three levels deep in this example:

```
sfo03-r1r5-core:Et6  sfo03-r1r8-sw1:Ethernet1
    sfo03-r1r8-sw1:Et22 sfo03-r6r8-sw3:Ethernet48
    sfo03-r1r8-sw1:Et24 sfo03-r6r8-sw2:Te1/0/1
    sfo03-r1r8-sw1:Et25 sfo03-r3r7-sw2:Te1/0/1
    sfo03-r1r8-sw1:Et26 sfo03-r3r7-sw1:24
```

It prints the port name next to the switch hostname, which allows you to see both "sides" of the inter-switch links. This is extremely useful when trying to orient yourself on the network. I'm still working on this script, but it currently produces a "real-time" network topology map that can be turned into a network diagram.

I hope this information inspires you to investigate network automation. Start with Netmiko Tools and the inventory file to get a sense of what is possible. You likely will encounter a scenario that requires some Python coding, either using the output of Netmiko Tools or perhaps your own standalone script. Either way, the Netmiko functions make automating a large, multivendor network fairly easy. ■

Eric Pearce is an IT Architect at Nutanix in San Jose, California. He has written for *Linux Journal* in the past, and he's also the author of several books published by O'Reilly and Associates.

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

IBM Began Buying Red Hat 20 Years Ago

How Big Blue became an open-source company.

By Glyn Moody

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+.

News that IBM is buying Red Hat is, of course, a significant moment for the world of free software. It's further proof, as if any were needed, that open source has won, and that even the mighty Big Blue must make its obeisance. Admittedly, the company is not quite the behemoth it was back in the 20th century, when "nobody ever got fired for buying IBM". But it remains a benchmark for serious, mainstream—and yes, slightly boring—computing. Its acquisition of Red Hat for the not inconsiderable sum of $34 billion, therefore, proves that selling free stuff is now regarded as a completely normal business model, acknowledged by even the most conservative corporations.

Many interesting analyses have been and will be written about why IBM bought Red Hat, and what it means for open source, Red Hat, Ubuntu, cloud computing, IBM, Microsoft and Amazon, amongst other things. But one aspect of the deal people may have missed is that in an important sense, IBM actually began buying Red Hat 20 years ago. After all, $34 billion acquisitions do not spring fully formed out of nowhere.

Reaching the point where IBM's management agreed it was the right thing to do required a journey. And, it was a particularly drawn-out and difficult journey, given IBM's starting point not just as the embodiment of traditional proprietary computing, but its very inventor.

Even the longest journey begins with a single step, and for IBM, it was taken on June 22, 1998. On that day, IBM announced it would ship the Apache web server with the IBM WebSphere Application Server, a key component of its WebSphere product family. Moreover, in an unprecedented move for the company, it would offer "commercial, enterprise-level support" for that free software.

When I was writing my book ***Rebel Code: inside Linux and the open source revolution*** in 2000, I had the good fortune to interview the key IBM employees who made that happen. The events of two years before still were fresh in their minds, and they explained to me why they decided to push IBM toward the bold strategy of adopting free software, which ultimately led to the company buying Red Hat 20 years later.

One of those people was James Barry, who was brought in to look at IBM's lineup in the web server sector. He found a mess there; IBM had around 50 products at the time. During his evaluation of IBM's strategy, he realized the central nature of the web server to all the other products. At that time, IBM's offering was Internet Connection Server, later re-branded to Domino Go. The problem was that IBM's web server held just 0.2% of the market; 90% of web servers came from Netscape (the first internet company, best known for its browser), Microsoft and Apache. Negligible market share meant it was difficult and expensive to find staff who were trained to use IBM's solution. That, in its turn, meant it was hard to sell IBM's WebSphere product line.

Barry, therefore, realized that IBM needed to adopt one of the mainstream web servers. IBM talked about buying Netscape. Had that happened, the history of open source would have been very different. As part of IBM, Netscape probably would not have released its browser code as the free software that became Mozilla. No Mozilla would have meant no Firefox, with all the knock-on effects that implies. But for

various reasons, the idea of buying Netscape didn't work out. Since Microsoft was too expensive to acquire, that left only one possibility: Apache.

For Barry, coming to that realization was easy. The hard part was convincing the rest of IBM that it was the right thing to do. He tried twice, unsuccessfully, to get his proposal adopted. Barry succeeded on the third occasion, in part because he teamed up with someone else at IBM who had independently come to the conclusion that Apache was the way forward for the company.

Shan Yen-Ping was working on IBM's e-business strategy in 1998 and, like Barry, realized that the web server was key in this space. Ditching IBM's own software in favor of open source was likely to be a traumatic experience for the company's engineers, who had invested so much in their own code. Shan's idea to request his senior developers to analyze Apache in detail proved key to winning their support. Shan says that when they started to dig deep into the code, they were surprised by the elegance of the architecture. As engineers, they had to admit that the open-source project was producing high-quality software. To cement that view, Shan asked Brian Behlendorf, one of the creators and leaders of the Apache project, to come in and talk with IBM's top web server architects. They too were impressed by him and his team's work. With the quality of Apache established, it was easier to win over IBM's developers for the move.

Shortly after the announcement that IBM would be adopting Apache as its web server, the company took another small but key step toward embracing open source more widely. It involved the Jikes Java compiler that had been written by two of IBM's researchers: Philippe Charles and Dave Shields. After a binary version of the program for GNU/Linux was released in July 1998, Shields started receiving requests for the source code. For IBM to provide access to the underlying code was unprecedented, but Shields said he would try to persuade his bosses that it would be a good move for the company.

A Jikes user suggested he should talk to Brian Behlendorf, who put him in touch with James Barry. IBM's recent adoption of Apache paved the way for Shield's own

efforts to release the company's code as open source. Shields wrote his proposal in August 1998, and it was accepted in September. The hardest part was not convincing management, but drawing up an open-source license. Shields said this involved "research attorneys, the attorneys at the software division who dealt with Java, the trademark attorneys, patents attorneys, contract attorneys". Everyone involved was aware that they were writing IBM's first open-source license, so getting it right was vital. In fact, the original Jikes license of December 1998 was later generalized into the IBM Public License in June 1999. It was a key moment, because it made releasing more IBM code as open source much easier, smoothing the way for the company's continuing march into the world of free software.

Barry described IBM as being like "a big elephant: very, very difficult to move an inch, but if you point the elephant toward the right direction and get it moving, it's also very difficult to stop it." The final nudge that set IBM moving inexorably toward the embrace of open source occurred on January, 10, 2000, when the company announced that it would make all of its server platforms "Linux-friendly", including the S/390 mainframe, the AS/400 minicomputer and the RS/6000 workstation. IBM was supporting GNU/Linux across its entire hardware range—a massive vote of confidence in freely available software written by a distributed community of coders.

The man who was appointed at the time was what amounted to a Linux Tsar for the company, Irving Wladawsky-Berger, said that there were three main strands to that historic decision. One was simply that GNU/Linux was a platform with a significant market share in the UNIX sector. Another was the early use of GNU/Linux by the supercomputing community something that eventually led to ***every single one of the world's top 500 supercomputers*** running some form of Linux today.

The third strand of thinking within IBM is perhaps the most interesting. Wladawsky-Berger pointed out how the rise of TCP/IP as the de facto standard for networking had made interconnection easy, and powered the rise of the internet and its astonishing expansion. People within IBM realized that GNU/Linux could do the same for application development. As he told me back in 2000:

> The whole notion separating application development from the underlying deployment platform has been a Holy Grail of the industry because it would all of the sudden unshackle the application developers from worrying about all that plumbing. I think with Linux we now have the best opportunity to do that. The fact that it's not owned by any one company, and that it's open source, is a huge part of what enables us to do that. If the answer had been, well, IBM has invented a new operating system, let's get everybody in the world to adopt it, you can imagine how far that would go with our competitors.

Far from inventing a "new operating system", with its purchase of Red Hat, IBM has now fully embraced the only one that matters any more—GNU/Linux. In doing so, it confirms Wladawsky-Berger's prescient analysis and completes that fascinating journey the company began all those years ago. ■

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