System76's Oryx Pro Laptop

FOSS Project Spotlight: Tutanota

How-To: Configure NAS

LINUX JOURNAL

Since 1994: The original magazine of the Linux community

PROGRAMMING

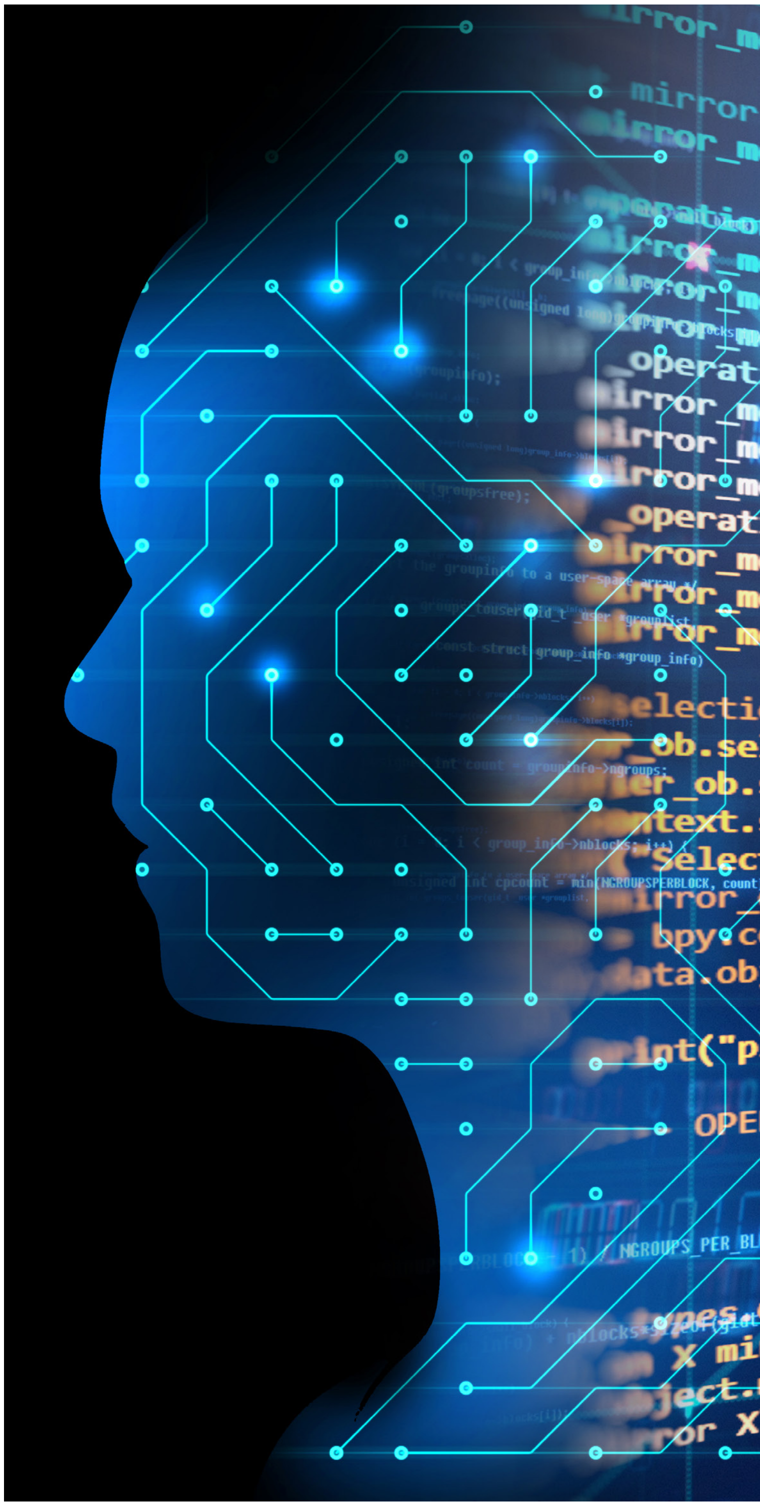
Understanding Bash | Getting Started with Rust
Intro to Go | Create Command-Line Tools with Clojure

ISSUE 291 | OCTOBER 2018
www.linuxjournal.com

CONTENTS

OCTOBER 2018
ISSUE 291

72 DEEP DIVE: Programming



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

ARTICLES

AT YOUR SERVICE

SUBSCRIPTIONS: *Linux Journal* is available as a digital magazine, in PDF, EPUB and MOBI formats. Renewing your subscription, changing your email address for issue delivery, paying your invoice, viewing your account details or other subscription inquiries can be done instantly online: http://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 http://www.linuxjournal.com/digital.

LETTERS TO THE EDITOR: We welcome your letters and encourage you to submit them at http://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: http://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 http://www.linuxjournal.com. Subscribe for free today: http://www.linuxjournal.com/enewsletters.

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

EXECUTIVE EDITOR: Jill Franklin, jill@linuxjournal.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

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

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!

Shall We Study Amazon's Pricing Together?

Is it possible to figure out how we're being profiled online?

By Doc Searls

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.

This past July, I spent a quality week getting rained out in a series of brainstorms by alpha data geeks at the Pacific Northwest BI & Analytics Summit in Rogue River, Oregon. Among the many things I failed to understand fully there was how much, or how well, we could know about how the commercial sites and services of the online world deal with us, based on what they gather about us, on the fly or over time, as we interact with them.

The short answer was "not much". But none of the experts I talked to said "Don't bother trying." On the contrary, the consensus was that the sums of data gathered by most companies are (in the words of one expert) "spaghetti balls" that are hard, if not possible, to unravel completely. More to my mission in life and work, they said it wouldn't hurt to have humans take some interest in the subject.

In fact, that was pretty much why I was invited there, as a Special Guest. My topic was "**When customers are in full command of what companies do with their data—and data about them**". As it says at that link, "The end of this story...is a new beginning for business, in a world where customers are fully in charge of their lives in the marketplace—both online and off: a world that was implicit in both the peer-to-peer design of the Internet and the nature of public markets in the pre-industrial world."

Obviously, this hasn't happened yet.

This became even more obvious during a break when I drove to our AirBnB nearby. By chance, my rental car radio was tuned to a program called **From Scurvy to Surgery: The History Of Randomized Trials**. It was an **Innovation Hub** interview with **Andrew Leigh**, Ph.D. (**@ALeighMP**), economist and member of the Australian Parliament, discussing his new book, ***Randomistas: How Radical Researchers Are Changing Our World*** (Yale University Press, 2018). At one point, Leigh reported that "One expert says, 'Every pixel on Amazon's home page has had to justify its existence through a randomized trial.'"

I thought, *Wow. How much of my own experience of Amazon has been as a randomized test subject? And can I possibly be in anything even remotely close to full charge of my own life inside Amazon's vast silo?*

After I got back to the meeting, I looked up Dr. Leigh's book on Amazon. (**Here's the link**.) I was kind of surprised to find the Kindle price was $26.12, while the hardcover price was the publisher's listed one: $27.50. Shouldn't the Kindle be a lot less? Then I found myself wondering: *Are these prices part of a randomized trial?*

So I conducted a quick trial of my own on two different browsers. One was Chrome, the other Brave. I wasn't logged in to Amazon on either, but on Brave, I did my best to mask where I was coming from by searching in a private tab over Tor. (Far as I know, Brave is the only browser to offer Tor on a start page.) I took screenshots of the results and posted them on Flickr, here. So you don't have to fire up a browser of your own, Table 1 shows the results.

Table 1. Results on July 27, 2018, in Rogue River

Book	Kindle	Hardcover	Paperback
Randomistas on Chrome	$26.12	$27.50	
Randomistas on Brave	$20.02	$27.50	
The Intention Economy on Chrome	$14.57	$14.50	
The Intention Economy on Brave	$19.65	$26.21	
The Cluetrain Manifesto (10th Anniversary Edition paperback) on Chrome	$11.99	$9.97	$13.11
The Cluetrain Manifesto (10th Anniversary Edition paperback) on Brave	$9.34	$9.97	$13.11
Biblio Tech on Chrome	$17.99	$13.93	
Biblio Tech on Brave	$14.04	$13.93	
Data and Goliath (hardcover) on Chrome		$21.30	
Data and Goliath (hardcover) on Brave		$21.26	
The Undoing Project on Chrome	$10.83	$9.46–$11.40	$13.75–$14.48
The Undoing Project on Brave	$9.90	$9.46–$11.40	$13.75–$14.48

Table 2. August 3, 2018, in New York City

Book	Kindle	Hardcover	Paperback
Randomistas on Chrome	$26.12	$27.50	
Randomistas on Brave	$32.66*	$27.50	
The Intention Economy on Chrome	$14.57	$14.50	
The Intention Economy on Brave	$18.21	$14.50	
The Cluetrain Manifesto (10th Anniversary Edition paperback) on Chrome	$11.99	$10.38	$9.89–$12.93***
The Cluetrain Manifesto (10th Anniversary Edition paperback) on Brave	$9.34	$10.38**	$13.11
Biblio Tech on Chrome	$17.99	$13.82	
Biblio Tech on Brave	$14.04	$13.82	
Data and Goliath (hardcover) on Chrome		$22.00****	
Data and Goliath (hardcover) on Brave		$22.00****	
The Undoing Project on Chrome	$10.39	$14.06	$14.48
The Undoing Project on Brave	$9.34	$14.06	$14.48

* Another price appeared and disappeared before I could read it. I also needed to answer a CAPTCHA to prove I wasn't a robot.

** Earlier it said "From $1.14".

*** Earlier it said "$9.89–$13.11".

**** Earlier it said "$21.26".

Since then, I've been checking Amazon prices often on different browsers, different devices, both logged in and not, and again using Tor on Brave. Prices for lots of stuff (not just books) have been all over the place. And information about products changes too.

For example, today (as I write this, on deadline) is August 19th. On Chrome, *Randomistas* is now $18.24 on Kindle, $19.20 in hardcover and available in paperback for $32.39. (It wasn't in paperback before. See **here**.) On Brave through Tor, it's the same for hardcover and paperback but $20.09 on Kindle.

To my amateur mind, Amazon's pricing calls to mind Dave Barry's classic column, "**The Unfriendly Skies**", published in 2010 and more relevant than ever today. Under "Answers to Common Airline Questions", he begins:

> **Q.** Airline fares are very confusing. How, exactly, does the airline determine the price of my ticket?
>
> **A.** Many cost factors are involved in flying an airplane from Point A to Point B, including distance, passenger load, whether each pilot will get his own pilot hat or they're going to share, and whether Point B has a runway.
>
> **Q.** So the airlines use these cost factors to calculate a rational price for my ticket?
>
> **A.** No. That is determined by Rudy the Fare Chicken, who decides the price of each ticket individually by pecking on a computer keyboard sprinkled with corn. If an airline agent tells you that they're having "computer problems," this means that Rudy is sick, and technicians are trying to activate the backup system, Conrad the Fare Hamster.

I now know, after doing some digging, that what's really going on here is "dynamic pricing", and there is a lot of jive about it on the web. (Here's a search: https://www.google.com/search?q=amazon+dynamic+pricing.) And I get that it's about lots of variables other than personal ones: A/B and randomized testing across populations, competitors' prices (again, viewed through different browsers or whatever Amazon's robots might use to simulate human queries), short- and long-term trends, inventory available now or back-up supply chains, scenarios, choice presentation and so on.

So here are a few serious questions: *How might we best research this from our side—the one where humans use browsers and actually buy stuff? Is it possible to figure out how we're being profiled, if at all—and how might we do that? Are there shortcuts to finding the cheapest Amazon price for a given product, among all the different prices it presents at different times and ways on different browsers, to persons logged in or not? Is this whole thing so opaque that we'll never know much more than a damn thing, and we're simply at the mercy of machines probing and manipulating us constantly?*

I'm hoping some of you have answers. I also think it would be fun to put together an Amazon Pricing System Research Project using *Linux Journal* readers. (Or maybe, hmm, Amazon's own **Mechanical Turk**.)

Possible? If so, write to me and let's see what we can do. ■

Note: pricing gun image used with this article courtesy of ***PriceGun.com****.*

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

LETTERS

Responses from Social Media

We asked the Twitterverse, "When did you begin using Linux on the desktop? What was your first distro?"

- **Alejandro LANGUREN Alex_Languren:** I started with a boxed edition of Turbolinux in 1999, now discontinued; the CDs are still in my personal library.

- **David Colon @davidpcolon:** Slackware in 1995, back when we had to create modelines in our XF86Config file.

- **Bill Studley @BillStudley:** Floppy version of Slackware that I downloaded from MIT

- **Jonathan Coker @jmc2038:** It was Slackware in 1994. Used most of a pack of 50 3.5 floppies to load Linux and XWin. Set up to dual boot with NT 3.51 (had to for a class) on a 1GB SCSI drive. Modified the settings for my system then recompiled the kernel. Still took less time than the Win install.

- **Mika Nieminen @peisi1:** My first was Linux Mandrake 10.0. Today Fedora, and tomorrow maybe openSUSE.

- **Kevin Adler @kadler_ibm:** Mandrake 7.2, around 2000–2001. Switched to @SUSE shortly thereafter (now @openSUSE), and it has been my primary OS since 2004. Windows-free since 2014 or so (I don't have definitive record that I can find).

- **Saptech @saptech:** Early 2000, Caldera Linux with KDE.

- **Rolf Besier @rolf_besier:** Jumped in first in 1996 with Delix, then SUSE. It was a pain in arse switching floppies like a disc jockey. Delix was for me state of the art in networking. Had some server experiences with SUN

Solaris & HP-UX for remote controlling. Now over 10 years on Debian with xfce4.

- **Daniel Bristot de Oliveira @bristot:** 2002, using @acmel's Conectiva! It came with KDE and 2.4.18 kernel. First I thought that the / would run with swap "FS" because it was the only FS listed with "linux" in the description. No internet in my small town in .br that time, CD bought with a magazine!

- **HomeTechHacker @HomeTechHacker:** Red Hat, I believe, back in 1995. Before that, I used NetBSD for a couple years. From 1999–2005 I used Windows, but I've been primarily Ubuntu since 6.06 in 2006, with some dabbles with Linux Mint, Kubuntu, Dreamlinux, Debian and Lubuntu.

- **Vincenzo Lalli @vincenzo_lalli:** It was 2007 I think. I bought a Dell that came with Win Vista. Soon I realized I had to find an alternative. I installed #Ubuntu 7-something. Over the years I tried many other distros, now mainly using #debian. Never looked back; it's been a liberating experience. #linux

- **Anders Karlsson @skallen:** 1993, Slackware downloaded floppies from http://funet.fi via a 9600 bps modem. It took all night to download them...

More from Social Media

Positive Internet @posipeople: The @linuxjournal is one of the few GNU/Linux institutions that's been around longer than us :-) Proper, in-depth technical analyses don't get any better than those in the journal. They merit all our support!

I am Gareth's Smirking Revenge @garethgreenaway: Finally got around to resubscribing to @linuxjournal and was pleasantly surprised to see that ALL previous issues were available digitally. How awesome is that. If you haven't already, you should go subscribe.

Photo of the Month

Andrea Polidori @andpoli: A relaxing moment with the @linuxjournal August issue on my #Linux based @pocketbook #eReader. That's cool. ;-) #ontheBeach

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

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

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

FOSS Project Spotlight: Tutanota, the First Encrypted Email Service with an App on F-Droid

Seven years ago, we started building Tutanota, an encrypted email service with a strong focus on security, privacy and open source. Long before the Snowden revelations, we felt there was a need for easy-to-use encryption that would allow everyone to communicate online without being snooped upon.

As developers, we know how easy it is to spy on email that travels through the web. Email, with its federated setup is great, and that's why it has become the main form of online communication and still is. However, from a security perspective, the federated setup is troublesome—to say the least.

End-to-end encrypted email is difficult to handle on desktops (with key generation, key sharing, secure storing of keys and so on), and it's close to impossible on mobile devices. For the average, not so tech-savvy internet user, there are a lot of pitfalls, and the probability of doing something wrong is, unfortunately, rather high.

That's why we decided to build Tutanota: a secure email service that is so easy to use, everyone can send confidential email, not only the tech-savvy. The entire

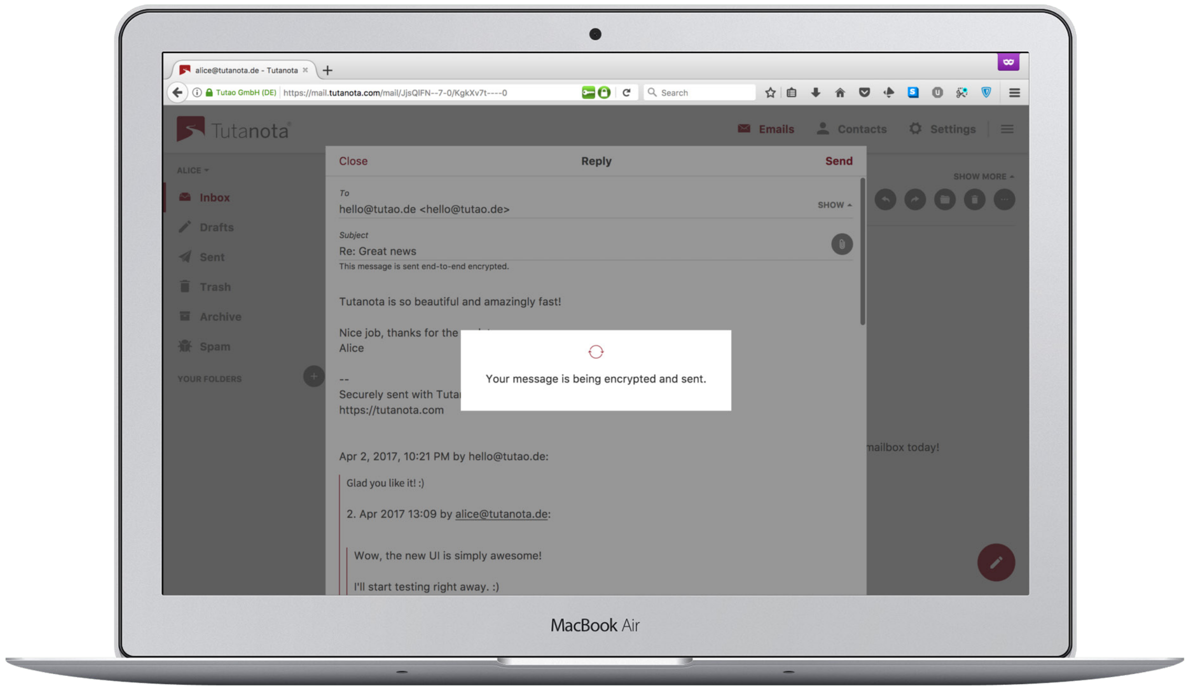
Figure 1. The Tutanota team's motto: "We fight for privacy with automatic encryption."

encryption process runs locally on users' devices, and it's fully automated. The automatic encryption also enabled us to build fully encrypted email apps for Android and iOS.

Finally, end-to-end encrypted email is starting to become the standard: 58% of all email sent from Tutanota already are end-to-end encrypted, and the percentage is constantly rising.

The Open-Source Email Service to Get Rid of Google

As open-source enthusiasts, our apps have been open source from the start, but



To
hello@tutao.de <hello@tutao.de>

Subject
Re: Great news
This message is sent end-to-end encrypted.

Tutanota is so beautiful and amazingly fast!

Nice job, thanks for the
Alice

--
Securely sent with Tuta
https://tutanota.com

Your message is being encrypted and sent.

Apr 2, 2017, 10:21 PM by hello@tutao.de:

Glad you like it! :)

2. Apr 2017 13:09 by alice@tutanota.de:

Wow, the new UI is simply awesome!

I'll start testing right away. :)

mailbox today!

Figure 2. Easy email encryption on desktops and mobile devices is now possible for everyone.

putting them on F-Droid was a challenge. As with all email services, we have used Google's FCM for push notifications. On top of that, our encrypted email service was based on Cordova, which the F-Droid servers are not able to build.

Not being able to publish our Android app on F-Droid was one of the main reasons we started to re-build the entire Tutanota web client. We are privacy and open-source enthusiasts; we ourselves use F-Droid. Consequently, we thought that our app must be published there, no matter the effort.

When rebuilding our email client, we made sure not to use Cordova anymore and to replace Google's FCM for push notifications.

The Challenge to Replace Google's FCM

GCM (or, as it's now called, FCM, for Firebase Cloud Messaging) is a service owned by Google. Unfortunately, FCM includes Google's tracking code for analytics purposes, which we didn't want to use. And, even more important: to use FCM, you have to send all your notification data to Google. You also have to use Google's proprietary libraries.

Because of privacy and security concerns, we didn't send any info in the notification messages. Therefore, the push notification mentioned only that you received a new message without a reference to the mailbox in which that message has been placed.

We wanted our users to be able to use Tutanota on every ROM and every device, without the control of a third-party. That's why we decided to take on the challenge and to build a push notification service ourselves.

When we started designing our push system, we set the following goals:

- It must be secure.
- It must be fast.
- It must be power-efficient.

We've researched how others (Signal, Wire, Conversations, Riot, Facebook and Mastodon) have been solving similar problems, and we had several options in mind, including WebSockets, MQTT, Server Sent Events and HTTP/2 Server Push.

We settled for the SSE (Server Sent Events), because it seemed like a simple solution. By that, I mean "easy to implement, easy to debug". Debugging these types of things can be a major headache, so one should not underestimate that factor. Another argument in favor of that solution was relative power efficiency. We didn't need upstream messages, and constant connection was not our goal.

So, What Is SSE?

SSE is a web API that allows a server to send events to connected clients. It's a relatively old API, which is, in my opinion, underused. We'd never heard of SSE before the federated network Mastodon, which uses SSE for real-time timeline updates, and it works great.

The protocol itself is very simple and resembles good old polling. The client opens a connection, and the server keeps it open. It's different from classical polling in that we keep this connection open for multiple events. The server can send events and data messages, they're just separated by new lines. So the only thing the client needs to do is to open a connection with a big timeout and read the stream in a loop.

SSE fits our needs better than WebSocket would (it's cheaper and converges faster, because it's not duplex). We've seen multiple chat apps trying to use WebSocket for push notifications, and it didn't seem power-efficient.

We had some experience with WebSocket already, and we knew that firewalls don't like keepalive connections. To solve this, we used the same workaround for SSE that we did for WebSocket. We sent "heartbeat" empty messages every few minutes. We made this interval adjustable from the server side and randomized it not to overwhelm the server.

In the end, we had to do some work—I could describe loads of challenges we had to overcome to make this finally work, but maybe some other time. Yet, it was totally worth it. Our new app is still in beta, but thanks to non-blocking IO, we've been able to maintain thousands of simultaneous connections without problems. Our users are no longer forced to use Google Play Services, and **we've been able to publish our app on F-Droid**.

As a side-note: wouldn't it be great if the user could just pick a "push notifications provider" in the phone settings and the OS managed all these hard details by itself, so every app that doesn't want to be policed by the platform owner didn't have to invent the system anew? It could be end-to-end encrypted between the app and the app server. There's no real technical difficulty in that, but as long as our systems are controlled by big players, we as app developers have to solve this by ourselves.

Tutanota Is the First App of an Email Service Available on F-Droid

Our app release on F-Droid really excites us, as it proves that it is possible to build a secure email service that's completely Google-free, giving people a real open-source alternative to the data-hungry market-leader Gmail.

This is a remarkable step, as so far no other email service has managed (or cared) to publish its app on F-Droid. The reason for this is that, in general, email services rely on Google's FCM for push notifications, which makes an F-Droid release impossible.

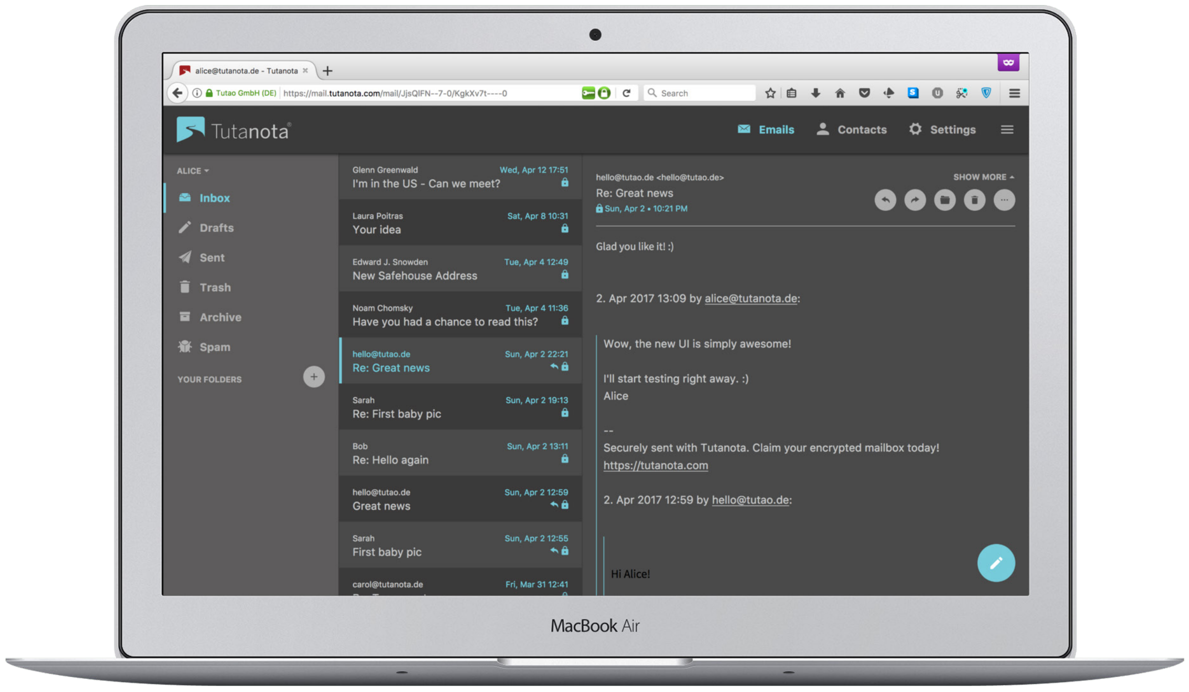
The F-Droid team also welcomed our move in the right direction:

> We are happy to see how enthusiastic Tutanota is about F-Droid and free software, having rewritten their app from scratch so it could be included. Furthermore, they take special measures to avoid tracking you, and the security looks solid with support for end-to-end encryption and two-factor authentication.

We are very excited about this release as well. And, we are thankful for the dedication and hard work of the numerous F-Droid volunteers helping us to publish our app there. We are also proud that the new Android app finally comes without any ties to Google services. As a secure email service, this is very important to us. We encourage our users to leave Google behind, so offering a Google-free Android app, therefore, is a minimum requirement for us.

A Privacy-Focused Email Service for Everyone

We've been using Tutanota ourselves for a couple years now. The new Tutanota client and apps are fast, come with a nice and minimalistic design, enable search on encrypted data, and support 2FA and auto-sync. Since we've added



Glenn Greenwald Wed, Apr 12 17:51
I'm in the US - Can we meet?

Laura Poitras Sat, Apr 8 10:31
Your idea

Edward J. Snowden Tue, Apr 4 12:49
New Safehouse Address

Noam Chomsky Tue, Apr 4 11:36
Have you had a chance to read this?

hello@tutao.de Sun, Apr 2 22:21
Re: Great news

Sarah Sun, Apr 2 19:13
Re: First baby pic

Bob Sun, Apr 2 13:11
Re: Hello again

hello@tutao.de Sun, Apr 2 12:59
Great news

Sarah Sun, Apr 2 12:55
First baby pic

carol@tutanota.de Fri, Mar 31 12:41

hello@tutao.de <hello@tutao.de>
Re: Great news
Sun, Apr 2 • 10:21 PM

Glad you like it! :)

2. Apr 2017 13:09 by alice@tutanota.de:

Wow, the new UI is simply awesome!

I'll start testing right away. :)
Alice

--
Securely sent with Tutanota. Claim your encrypted mailbox today!
https://tutanota.com

2. Apr 2017 12:59 by hello@tutao.de:

Hi Alice!

Figure 3. The new Tutanota client comes with a dark theme—a nice and minimalistic design that lets you easily encrypt email messages to every email address in the world.

search, there's no major feature missing for professional use any longer, and we've noticed the numbers of new users rising constantly. We recommend that everyone who wants to stop third parties from reading their private email to just give it a try.

—Matthias Pfau

Patreon and *Linux Journal*

PATREON

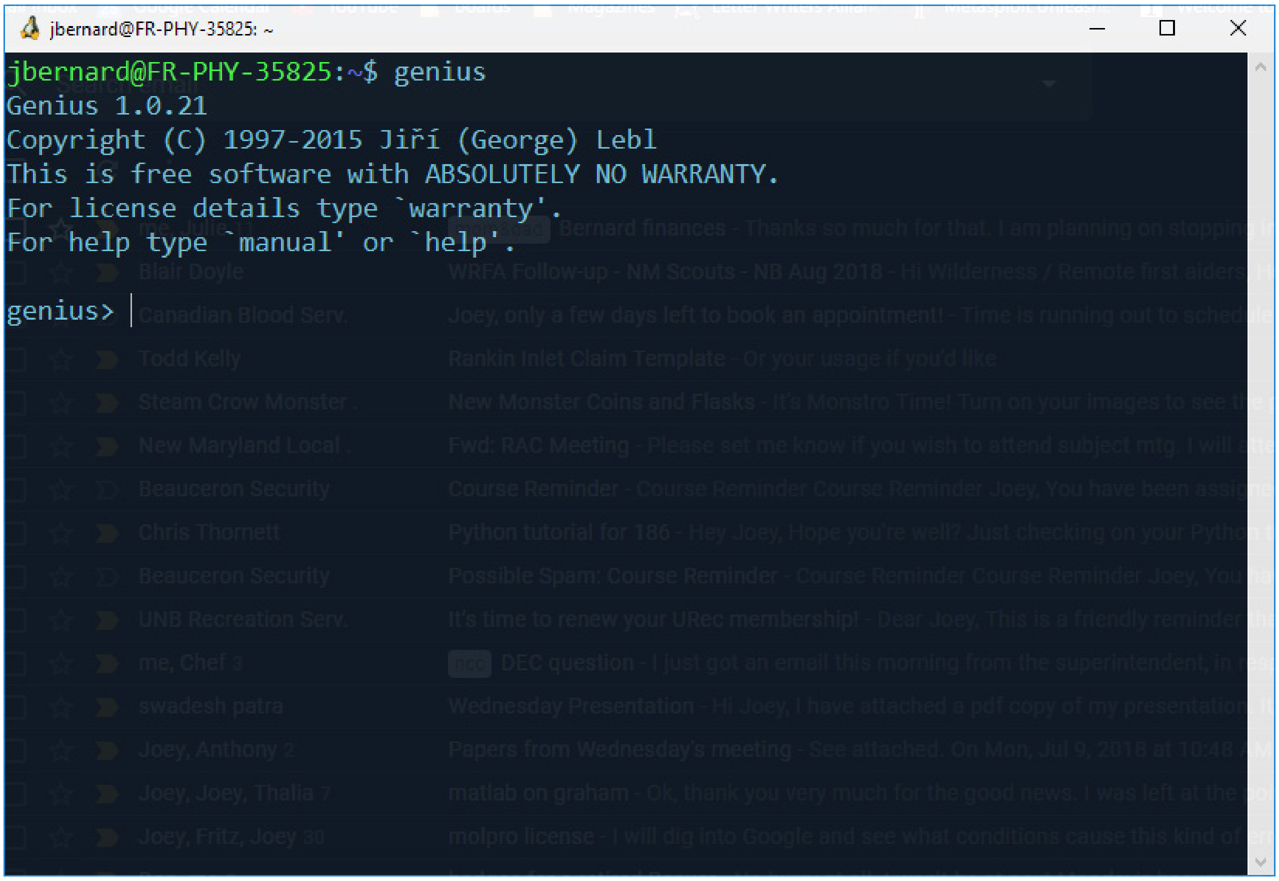
Together with the help of *Linux Journal* supporters and subscribers, we can offer trusted reporting for the world of open-source today, tomorrow and in the future. To our subscribers, old and new, we sincerely thank you for your continued support. In addition to magazine subscriptions, we are now receiving 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
- David Breakey
- Dr. Stuart Makowski
- James Mayes
- James Weatherell
- Josh Simmons
- Magnus Magicman
- Mostly_Linux
- NDCHost.com
- Robert J. Hansen

BECOME A PATRON

Introducing Genius, the Advanced Scientific Calculator for Linux

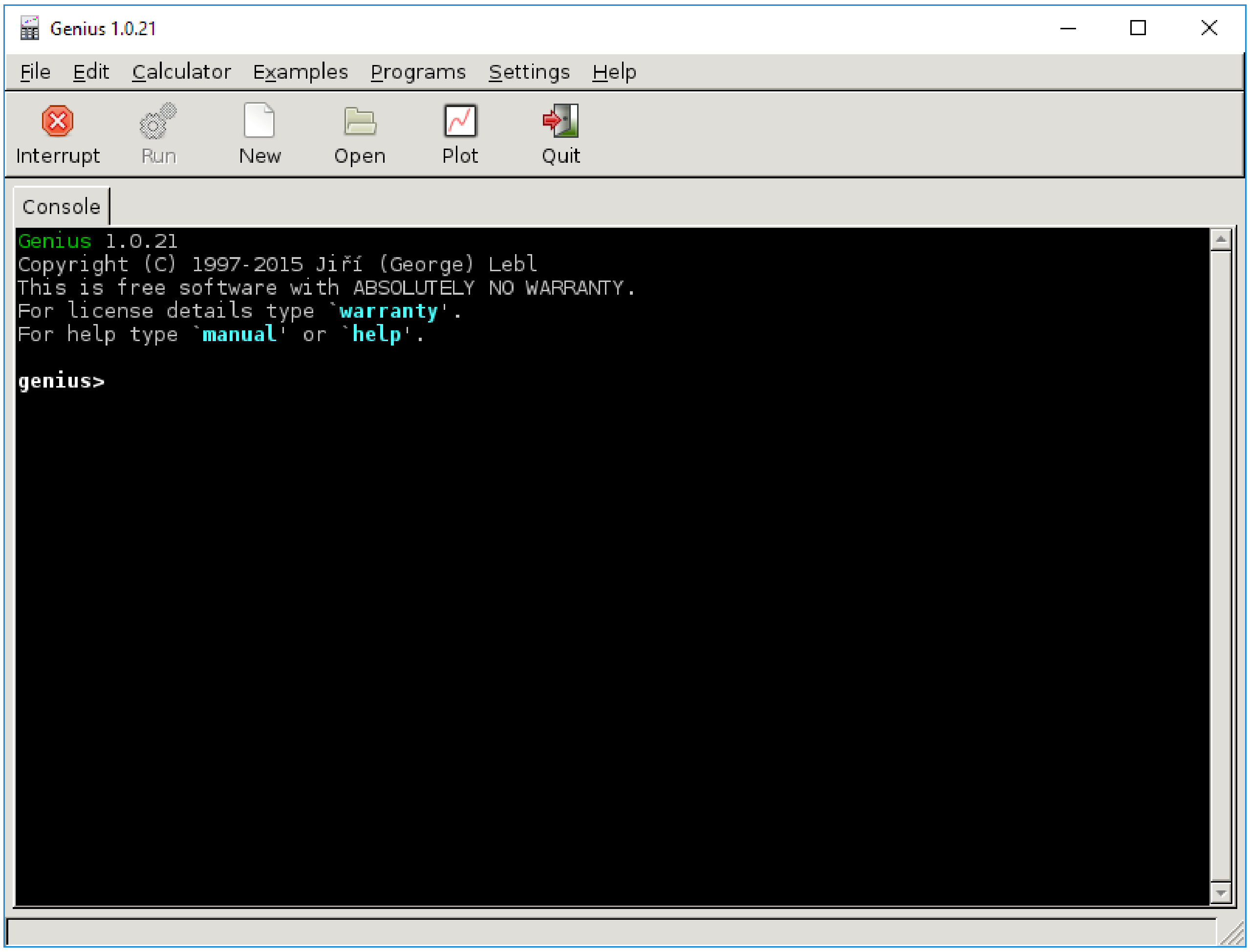
Genius is a calculator program that has both a command-line version and a GNOME GUI version. It should available in your distribution's package management system. For Debian-based distributions, the GUI version and the command-line version are



```
jbernard@FR-PHY-35825:~$ genius
Genius 1.0.21
Copyright (C) 1997-2015 Jiří (George) Lebl
This is free software with ABSOLUTELY NO WARRANTY.
For license details type `warranty'.
For help type `manual' or `help'.

genius>
```

Figure 1. When you start Genius, you get the version and some license information, and then you'll see the interpreter prompt.



```
Genius 1.0.21
Copyright (C) 1997-2015 Jiří (George) Lebl
This is free software with ABSOLUTELY NO WARRANTY.
For license details type `warranty'.
For help type `manual' or `help'.

genius>
```

Figure 2. The GUI interface provides easy menu access to most of the functionality within Genius.

two separate packages. Assuming that you want to install both, you can do so with the following command:

```
sudo apt-get install genius gnome-genius
```

If you use Ubuntu, be aware that the package gnome-genius doesn't appear to be in Bionic. It's in earlier versions (trusty, xenial and arty), and it appears to be in the next version (cosmic). I ran into this problem, and thought I'd mention it to save you some aggravation.

Starting the command-line version provides an interpreter that should be familiar to Python or R users.

If you start gnome-genius, you'll see a graphical interface that is likely to be more comfortable to new users. For the rest of this article, I'm using the GUI version in order to demonstrate some of the things you can do with Genius.

You can use Genius just as a general-purpose calculator, so you can do things like:

```
genius> 4+5
= 9
```

Along with basic math operators, you also can use trigonometric functions. This command gives the sine of 45 degrees:

```
genius> sin(45)
= 0.850903524534
```

These types of calculations can be of essentially arbitrary size. You also can use complex numbers out of the box. Many other standard mathematical functions are available as well, including items like logarithms, statistics, combinatorics and even calculus functions.

Along with functions, Genius also provides control structures like conditionals and looping structures. For example, the following code gives you a basic `for` loop that prints out the sine of the first 90 degrees:

```
for i = 1 to 90 do (
   x = sin(i);
   print(x)
)
```

As you can see, the syntax is almost C-like. At first blush, it looks like the semicolon is being used as a line-ending character, but it's actually a command separator. That's why there is a semicolon on the line with the sine function, but there is no semicolon on the line with the print function. This means you could write the `for` loop as the following:

```
for i = 1 to 90 do ( x = sin(i); print(x) )
```

Along with `for` loops, there are `while` loops, `until` loops, `do-while` loops, `do-until` loops and `foreach` loops. You also can control whether or not to pop out of a loop with the `break` and `continue` commands. They behave the same way that they do in languages like C. The conditional structure in Genius is a very basic `if` structure, so a basic `if-then` statement looks like the following:

```
if (a==5) then (a=a-1)
```

You also can use an `else` statement:

```
if (c>0) then (c=c-1) else (c=0)
```

Genius has no `elseif` statement.

You can use conditionals anywhere you would put an expression, which means you could use an `if` structure to set a variable value:

```
a = (if b>0 then b else 1)
```

As you can see, I didn't use parentheses here. You need to use them only in cases where the order of operations might be confusing.

So far, I've covered commands, variables, conditionals and looping structures. Genius also claims it uses a programming language called GEL. A programming language should have one last structure, the ability to organize code into reusable chunks. And, of course, GEL has the ability for end users to define their own functions. The basic syntax of a function definition looks like this:

```
function <identifier>(<comma separated arguments>) =
 ↪<function body>
```

As a really simple example, the following code defines a cubing function:

```
function my_cube(x) = x*x*x
```

You then can use it just like any other function:

```
genius> my_cube(3) = 27
```

Sometimes, you may need to be able to handle a variable list of input parameters to your function. In those cases, you define your function with the last parameter being "...". It looks like the following:

```
function my_func(a, b, c...) = <function body>
```

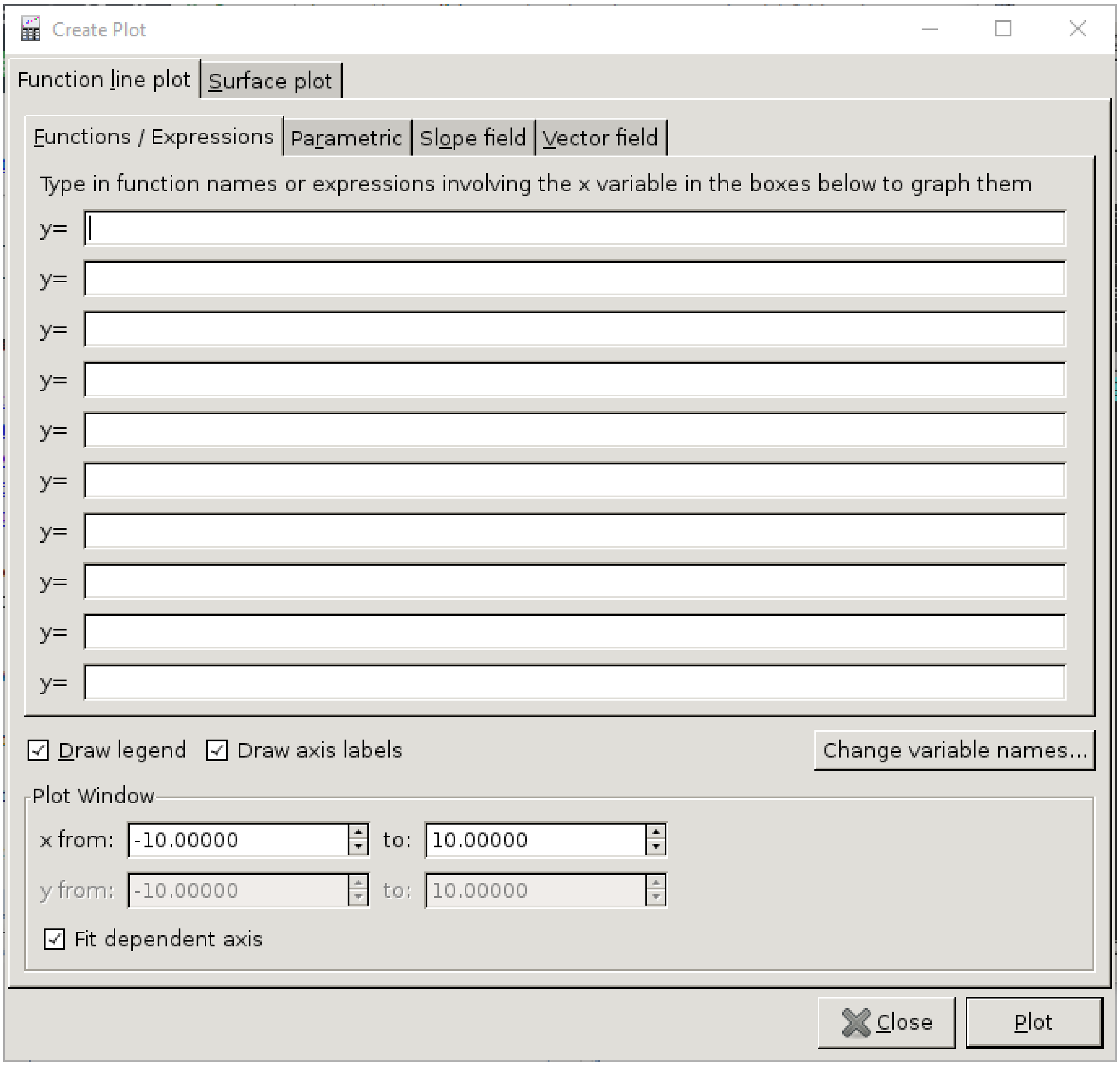
In such cases, the input parameters are handed to your function body as a vector of values.

When you start writing larger pieces of code, you likely will need to start handling error conditions. Genius (and, hence, GEL) has basic error handling capabilities. When you detect an error in your code, you can send a message to the end user with the `error` command:

```
if not IsMatrix (M) then (
   error("M is not a matrix")
)
```

This might not be enough, however. If the error isn't recoverable, you'll need to stop execution somehow. GEL provides two options. The first is to stop the current function and go back to the calling code using the `bailout` command. If the error is extremely horrendous, you may need to stop all execution. In those cases, you can use the `exception` command.

Genius also has a huge number of advanced functions. As an example of the kinds of advanced calculations you can do, let's look at doing a numerical integration.



Type in function names or expressions involving the x variable in the boxes below to graph them

y=
y=
y=
y=
y=
y=
y=
y=
y=
y=

☑ Draw legend ☑ Draw axis labels

Plot Window

x from: -10.00000 to: 10.00000

y from: -10.00000 to: 10.00000

☑ Fit dependent axis

Figure 3. The "Create Plot" window lets you define both line plots and surface plots for multiple functions.

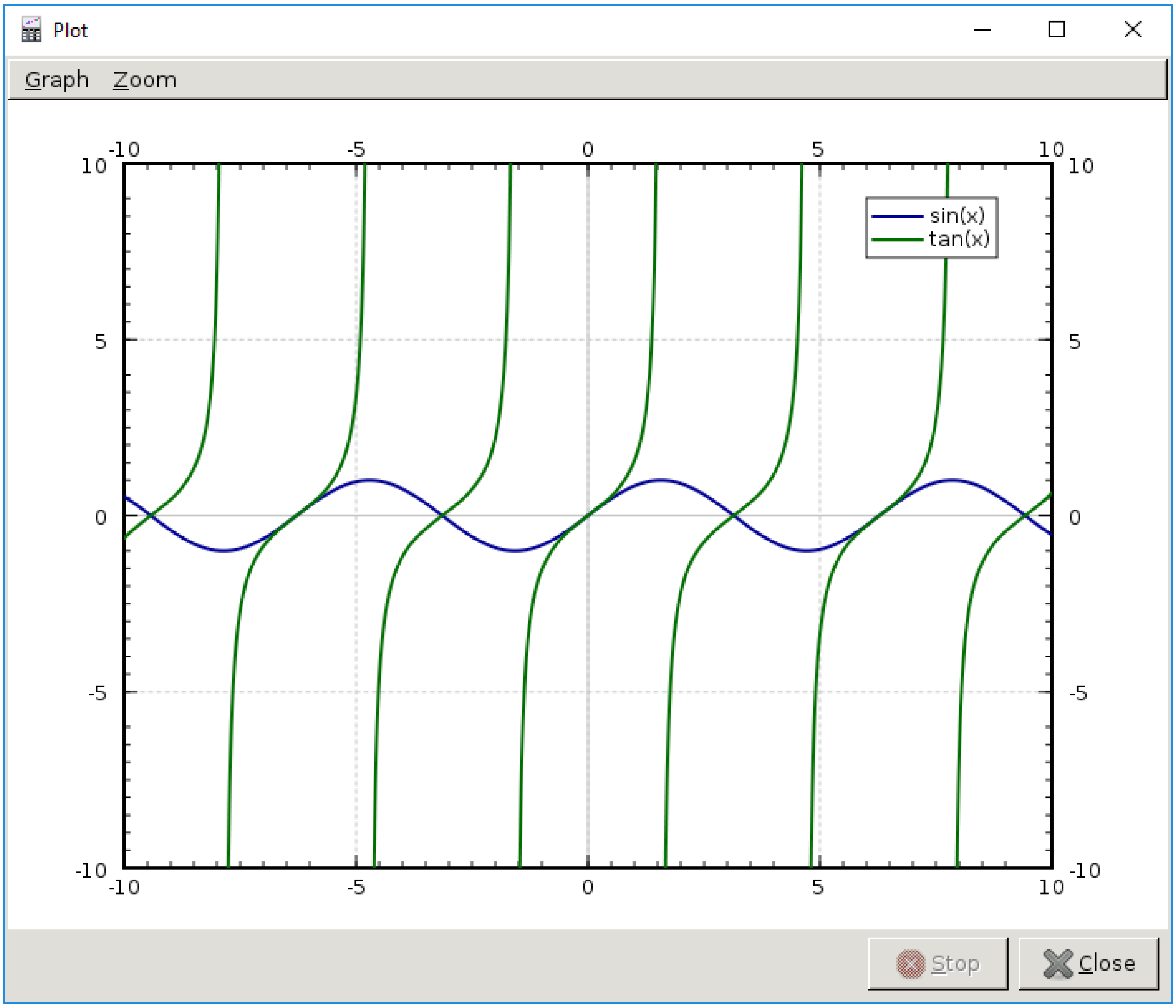


Figure 4. GNOME Genius lets you plot multiple functions easily. For example, you could plot sine and tangent in order to see how they compare to each other.

You can integrate a function, from a start limit to an end limit. For example, you can find the numerical integral of the sine function from 0 degrees to 180 degrees with the following:

```
genius> NumericalIntegral(sin, 0, 180) = 1.59846942736
```

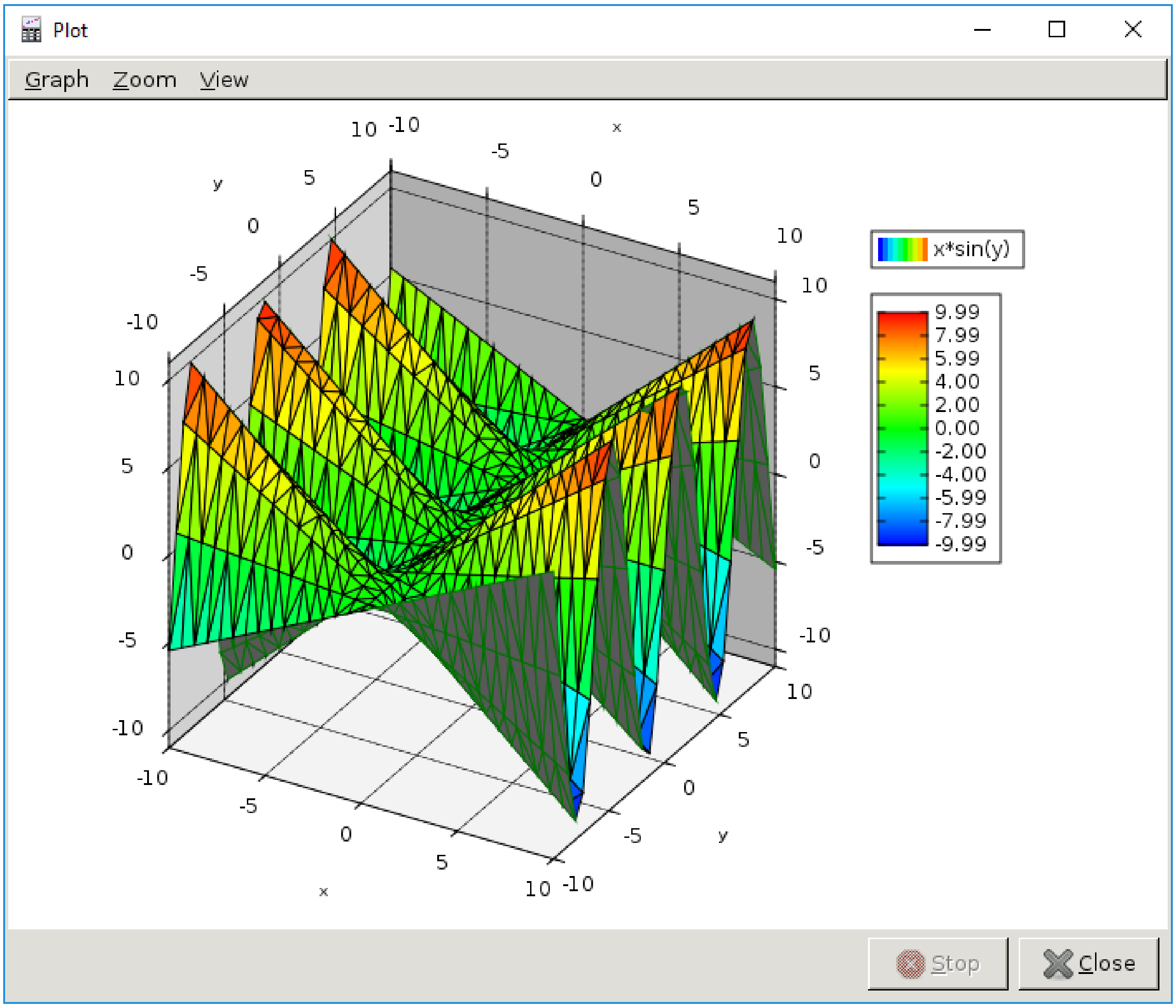


Figure 5. You easily can plot a single function in terms of x and y—for example, x*sin(y).

You also can do infinite sums, numerical derivatives and limits.

The last item I want to look at is available only with the GNOME version of Genius. In this case, you have the ability to plot data and functions and display them on the screen. When you click on the plot button on the main window, you'll get a new window where you can define the plot parameters.

Since you can plot multiple functions, you can see them side by side in the same window. If, instead, you need to do a 3D plot of a surface, you can select the surface plot tab of the plotting window and define a function in terms of x and y. Within the plot window, there are several options for changing the view. For the surface plot, you even can make it rotate so you can see the resultant plot from all angles. When you have the plot looking exactly the way you want, click the Graph menu entry and export it to one of several file formats so you can use it in other publications or reports.

As you can see, Genius provides a fair bit of functionality within a small package. It's been used in education to allow students to see the results of different calculations quickly and to show how they vary based on inputs or changes in algorithm. As well, it provides the essentials of an advanced scientific calculator. People who have used the HP or TI advanced hand-held calculators will find Genius a very powerful replacement on the desktop. You can find much more information at the main website, including the manual and a set of examples.

—Joey Bernard

News Briefs

Visit LinuxJournal.com for daily news briefs.

- Mozilla announced its 2018–2019 Fellows in openness, science and tech policy. These fellows "will spend the next 10 to 12 months creating a more secure, inclusive, and decentralized internet". In the past, Mozilla fellows "built secure platforms for LGBTQ individuals in the Middle East; leveraged open-source data and tools to bolster biomedical research across the African continent; and raised awareness about invasive online tracking." See the Mozilla blog for more information and the list of Fellows.

- According to a recent Cloud Foundry Foundation (CFF) survey, Java and JavaScript are the top enterprise languages. See ZDNet for more information on the survey results.

- Valve announced that it's releasing the Beta of a new and improved Steam Play version to Linux. The new version includes "a modified distribution of Wine, called Proton, to provide compatibility with Windows game titles." Other improvements include DirectX 11 and 12 implementations are now based on Vulkan, full-screen support has been improved, game controller support has been improved, and "Windows games with no Linux version currently available can now be installed and run directly from the Linux Steam client, complete with native Steamworks and OpenVR support".

- Intel has now reworked the license for its microcode security fix after outcry from the community. The Register quotes Imad Sousou, corporate VP and general manager of Intel Open Source Technology Center, "We have simplified the Intel license to make it easier to distribute CPU microcode updates and posted the new version here. As an active member of the open source community, we continue to welcome all feedback and thank the community."

- UBports Foundation has released Ubuntu Touch OTA-4. This release features Ubuntu 16.04 and includes many security fixes and stability improvements.

UBports notes that "We believe that this is the 'official' starting point of the UBports project. From the point when Canonical dropped the project until today, the community has been playing 'catch up' in development, infrastructure, and community building. This release shows that the community is soundly based and capable of delivering."

- NordVPN recently released the NordVPN Linux app. This dedicated app for Linux makes it even easier to install the VPN on your machine. For more information and to download, visit the NordVPN for Linux download page.

- Mozilla announced a different approach to anti-tracking on the internet. Mozilla's new approach means that "in the near future, Firefox will—by default—protect users by blocking tracking while also offering a clear set of controls to give our users more choice over what information they share with sites." In order to accomplish this, Mozilla has three key initiatives: improve page load performance, remove cross-site tracking and mitigate harmful practices.

- Linux Mint Debian Edition "Cindy" is now available. LDME's goal is to be as similar as possible to Linux Mint, but with a Debian base instead of Ubuntu. See the release notes for more information.

- Tor Browser version 8.0 was released. This is the first stable release based on Firefox 60 ESR, and it includes "a new user onboarding experience; an updated landing page that follows our styleguide; additional language support; and new behaviors for bridge fetching, displaying a circuit, and visiting .onion sites." You can download it from here.

- Nextcloud announced the release of version 14. This new version introduces two big security improvements: video verification and signal/telegram/SMS 2FA support. Version 14 also includes many collaboration improvements as well as a Data Protection Confirmation app in compliance with the GDPR. Go here to install.

Thanks to Sponsors **Linode** and **Pulseway** for Supporting *Linux Journal*

linode

Cloud Hosting
for You.

High performance SSD Linux servers for all of your infrastructure needs.

www.linode.com

System Management
at Your Fingertips.

www.pulseway.com

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

Papa's Got a Brand New NAS: the Software

Who needs a custom NAS OS or a web-based GUI when command-line NAS software is so easy to configure?

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.

In a recent letter to the editor, I was contacted by a reader who enjoyed my "Papa's Got a Brand New NAS" article, but wished I had spent more time describing the software I used. When I wrote the article, I decided not to dive into the software too much, because it all was pretty standard for serving files under Linux. But on second thought, if you want to re-create what I made, I imagine it would be nice to know the software side as well, so this article describes the software I use in my home NAS.

The OS

My NAS uses the ODROID-XU4 as the main computing platform, and so far, I've found its octo-core ARM CPU and the rest of its resources to be adequate for a home NAS. When I first set it up, I visited the official wiki page for the computer, which provides a number of OS images, including Ubuntu and Android images that you can copy onto a microSD card. Those images are geared more toward desktop use, however, and I wanted a minimal server image. After some searching, I found a minimal image for what was the current Debian stable release at the time (Jessie).

Although this minimal image worked okay for me, I don't necessarily recommend just going with whatever OS some volunteer on a forum creates. Since I first set up the computer, the Armbian project has been released, and it supports a number of standardized OS images for quite a few ARM platforms including the ODROID-XU4. So if you want to follow in my footsteps, you may want to start with the minimal Armbian Debian image.

If you've ever used a Raspberry Pi before, the process of setting up an alternative ARM board shouldn't be too different. Use another computer to write an OS image to a microSD card, boot the ARM board, and at boot, the image will expand to fill the existing filesystem. Then reboot and connect to the network, so you can log in with the default credentials your particular image sets up. Like with Raspbian builds, the first step you should perform with Armbian or any other OS image is to change the default password to something else. Even better, you should consider setting up proper user accounts instead of relying on the default.

The nice thing about these Debian-based ARM images is that you end up with a kernel that works with your hardware, but you also have the wide variety of software that Debian is known for at your disposal. In general, you can treat this custom board like any other Debian server. I've been using Debian servers for years, and many online guides describe how to set up servers under Debian, so it provides a nice base platform for just about anything you'd like to do with the server.

In my case, since I was migrating to this new NAS from an existing 1U Debian server, including just moving over the physical hard drives to a new enclosure, the fact that the distribution was the same meant that as long as I made sure I installed the same packages on this new computer, I could generally just copy over my configuration files wholesale from the old computer. This is one of the big benefits to rolling your own NAS off a standard Linux distribution instead of using some prepackaged NAS image. The prepackaged solution may be easier at first, but if you ever want to migrate off of it to some other OS, it may be difficult, if not impossible, to take advantage of any existing settings. In my situation, even if I had gone with another Linux distribution, I still could have copied over all of my configuration files to the new distribution—in some cases even into the same exact directories.

NFS

As I mentioned, since I was moving from an existing 1U NAS server built on top of standard Debian services, setting up my NFS service was a simple matter of installing the nfs-kernel-server Debian package, copying my /etc/exports file over from my old server and restarting the nfs-kernel-server service with:

```
$ sudo service nfs-kernel-server restart
```

If you're not familiar with setting up a traditional NFS server under Linux, so many different guides exist that I doubt I'd be adding much to the world of NFS documentation by rehashing it again here. Suffice it to say that it comes down to adding entries into your /etc/exports file that tell the NFS server which directories to share, who to share them with (based on IP) and what restrictions to use. For instance, here's a sample entry I use to share a particular backup archive directory with a particular computer on my network:

```
/mnt/storage/archive 192.168.0.50(fsid=715,rw)
```

This line tells the NFS server to share the local /mnt/storage/archive directory with the machine that has the IP 192.168.0.50, to give it read/write privileges and also to assign this particular share with a certain filesystem ID. I've discovered that assigning a unique `fsid` value to each entry in /etc/exports can help the NFS server identify each filesystem it's exporting explicitly with this ID, in case it can't find a UUID for the filesystem (or if you are exporting multiple directories within the same filesystem). Once I make a change to the /etc/exports file, I like to tell the NFS service to reload the file explicitly with:

```
$ sudo service nfs-kernel-server reload
```

NFS has a lot of different and complicated options you can apply to filesystems, and there's a bit of an art to tuning things exactly how you want them to be (especially if you are deciding between version 3 and 4 of the NFS protocol). I typically turn to the exports man page (type `man exports` in a terminal) for good descriptions of all the options and to see configuration examples.

Samba

If you just need to share files with Linux clients, NFS may be all you need. However, if you have other OSes on your network, or clients who don't have good NFS support, you may find it useful to offer Windows-style SMB/CIFS file sharing using Samba as well. Although Samba is configured quite differently from NFS, it's still not too complicated.

First, install the Samba package for your distribution. In my case, that meant:

```
$ sudo apt install samba
```

Once the package is installed, you will see that Debian provides a well commented /etc/samba/smb.conf file with ordinary defaults set. I then edited that /etc/samba/smb.conf file and made sure to restrict access to my Samba service to only those IPs I wanted to allow by setting the following options in the networking section of the smb.conf:

```
hosts allow = 192.168.0.20, 192.168.0.22, 192.168.0.23
interfaces = 127.0.0.1 192.168.0.1/24
bind interfaces only = Yes
```

These changes restrict Samba access to only a few IPs, and explicitly tell Samba to listen to localhost and a particular interface on the correct IP network.

There are additional ways you can configure access control with Samba, and by default, Debian sets it up so that Samba uses local UNIX accounts. This means you can set up local UNIX accounts on the server, give them a strong password, and then require that users authenticate with the appropriate user name and password before they have access to a file share. Because this is already set up in Debian, all I had left to do was to add some file shares to the end of my smb.conf file using the commented examples as a reference. This example shows how to share the same /mnt/storage/archive directory with Samba instead of NFS:

```
[archive]
  path = /mnt/storage/archive/
  revalidate = Yes
  writeable = Yes
  guest ok = No
  force user = greenfly
```

As with NFS, there are countless guides on how to configure Samba. In addition to those guides, you can do as I do and check out the heavily commented smb.conf or type `man smb.conf` if you want more specifics on what a particular option does. As with NFS, when you change a setting in smb.conf, you need to reload Samba with:

```
$ sudo service samba reload
```

Conclusion

What's refreshing about setting up Linux as a NAS is that file sharing (in particular, replacing Windows SMB file servers in corporate environments) is one of the first major forays Linux made in the enterprise. As a result, as you have seen, setting up Linux to be a NAS is pretty straightforward even without some nice GUI. What's more, since I'm just using a normal Linux distribution instead of some custom NAS-specific OS, I also can use this same server for all sorts of other things, such as a local DNS resolver, local mail relay or any other Linux service I might think of. Plus, down the road if I ever feel a need to upgrade, it should be pretty easy to move these configurations over to brand new hardware. ■

Resources

"Papa's Got a Brand New NAS" by Kyle Rankin, *LJ*, September, 2016

ODROID-XU4

Official Wiki Page for ODROID-XU4

Original Minimal Jessie Image for Odroid XU4

Armbian Images

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

Automate Sysadmin Tasks with Python's os.walk Function

Using Python's os.walk function to walk through a tree of files and directories.

By Reuven M. Lerner

Reuven M. Lerner teaches Python, data science and Git to companies around the world. A new cohort of his online "Weekly Python Exercise" course will be starting in late September 2018; learn more at http://WeeklyPythonExercise.com. You also can subscribe to his free, weekly "better developers" email list and the courses he offers at http://lerner.co.il. Reuven lives with his wife and children in Modi'in, Israel.

I'm a web guy; I put together my first site in early 1993. And so, when I started to do Python training, I assumed that most of my students also were going to be web developers or aspiring web developers. Nothing could be further from the truth. Although some of my students certainly are interested in web applications, the majority of them are software engineers, testers, data scientists and system administrators.

This last group, the system administrators, usually comes into my course with the same story. The company they work for has been writing Bash scripts for several years, but they want to move to a higher-level language with greater expressiveness and a large number of third-party add-ons. (No offense to Bash users is intended; you can do amazing things with Bash, but I hope you'll agree that the scripts can become unwieldy and hard to maintain.)

It turns out that with a few simple tools and ideas, these system administrators can use Python to do more with less code, as well as create reports and maintain servers. So in this article, I describe one particularly useful tool that's often overlooked: os.walk, a function that lets you walk through a tree of files and directories.

os.walk Basics

Linux users are used to the `ls` command to get a list of files in a directory. Python comes with two different functions that can return the list of files. One is `os.listdir`, which means the "listdir" function in the "os" package. If you want, you can pass the name of a directory to `os.listdir`. If you don't do that, you'll get the names of files in the current directory. So, you can say:

```
In [10]: import os
```

When I do that on my computer, in the current directory, I get the following:

```
In [11]: os.listdir('.')
Out[11]:
['.git',
 '.gitignore',
 '.ipynb_checkpoints',
 '.mypy_cache',
 'Archive',
 'Files']
```

As you can see, `os.listdir` returns a list of strings, with each string being a filename. Of course, in UNIX-type systems, directories are files too—so along with files, you'll also see subdirectories without any obvious indication of which is which.

I gave up on `os.listdir` long ago, in favor of `glob.glob`, which means the "glob" function in the "glob" module. Command-line users are used to using "globbing", although they often don't know its name. Globbing means using the * and ? characters, among others, for more flexible matching of filenames. Although `os.listdir` can return the list of files in a directory, it cannot filter them. You can though with `glob.glob`:

```
In [13]: import glob

In [14]: glob.glob('Files/*.zip')
Out[14]:
['Files/advanced-exercise-files.zip',
 'Files/exercise-files.zip',
 'Files/names.zip',
 'Files/words.zip']
```

In either case, you get the names of the files (and subdirectories) as strings. You then can use a `for` loop or a list comprehension to iterate over them and perform an action. Also note that in contrast with `os.listdir`, which returns the list of filenames without any path, `glob.glob` returns the full pathname of each file, something I've often found to be useful.

But what if you want to go through each file, including every file in every subdirectory? Then you have a bit more of a problem. Sure, you could use a `for` loop to iterate over each filename and then use `os.path.isdir` to figure out whether it's a subdirectory—and if so, then you could get the list of files in that subdirectory and add them to the list over which you're iterating.

Or, you can use the `os.walk` function, which does all of this and more. Although `os.walk` looks and acts like a function, it's actually a "generator function"—a function that, when executed, returns a "generator" object that implements the iteration protocol. If you're not used to working with generators, running the function can be a bit surprising:

```
In [15]: os.walk('.')
Out[15]: <generator object walk at 0x1035be5e8>
```

The idea is that you'll put the output from `os.walk` in a `for` loop. Let's do that:

```
In [17]: for item in os.walk('.'):
    ...:     print(item)
```

The result, at least on my computer, is a huge amount of output, scrolling by so fast that I can't read it easily. Whether that happens to you depends on where you run this `for` loop on your system and how many files (and subdirectories) exist.

In each iteration, `os.walk` returns a tuple containing three elements:

- The current path (that is, directory name) as a string.
- A list of subdirectory names (as strings).
- A list of non-directory filenames (as strings).

So, it's typical to invoke `os.walk` such that each of these three elements is assigned to a separate variable in the `for` loop:

```
In [19]: for currentdir, dirnames, filenames in os.walk('.'):
    ...:     print(currentdir)
```

The iterations continue until each of the subdirectories under the argument to `os.walk` has been returned. This allows you to perform all sorts of reports and interesting tasks. For example, the above code will print all of the subdirectories under the current directory, ".".

Counting Files

Let's say you want to count the number of files (not subdirectories) under the current directory. You can say:

```
In [19]: file_count = 0
In [20]: for currentdir, dirnames, filenames in os.walk('.'):
    ...:     file_count += len(filenames)
    ...:

In [21]: file_count
Out[21]: 3657
```

You also can do something a bit more sophisticated, counting how many files there are of each type, using the extension as a classifier. You can get the extension with `os.path.splitext`, which returns two items—the filename without the extension and the extension itself:

```
In [23]: os.path.splitext('abc/def/ghi.jkl')
Out[23]: ('abc/def/ghi', '.jkl')
```

You can count the items using one of my favorite Python data structures, `Counter`. For example:

```
In [24]: from collections import Counter

In [25]: counts = Counter()

In [26]: for currentdir, dirnames, filenames in os.walk('.'):
    ...:     for one_filename in filenames:
    ...:         first_part, ext =
 ↪os.path.splitext(one_filename)
    ...:         counts[ext] += 1
```

This goes through each directory under ".", getting the filenames. It then iterates through the list of filenames, splitting the name so that you can get the extension. You then add 1 to the counter for that extension.

Once this code has run, you can ask `counts` for a report. Because it's a dict, you can use the `items` method and print the keys and values (that is, extensions and counts). You can print them as follows:

```
In [30]: for extension, count in counts.items():
    ...:     print(f"{extension:8}{count}")
```

In the above code, `f strings` displays the extension (in a field of eight characters) and the count.

Wouldn't it be nice though to show only the ten most common extensions? Yes, but then you'd have to sort through the `counts` object. It's much easier just to use the `most_common` method that the `Counter` object provides, which returns not only the keys and values, but also sorts them in descending order:

```
In [31]: for extension, count in counts.most_common(10):
    ...:     print(f"{extension:8}{count}")
    ...:
.py     1149
        867
.zip    466
.ipynb  410
.pyc    372
.txt    151
.json   76
.so     37
.conf   19
.py~    12
```

In other words—not surprisingly—this example shows that the most common file extension in the directory I use for teaching Python courses is .py. Files without any extension are next, followed by .zip, .ipynb (Jupyter notebooks) and .pyc (byte-compiled Python).

File Sizes

You can ask more interesting questions as well. For example, perhaps you want to know how much disk space is used by each of these file types. Now you don't add 1 for each time you encounter a file extension, but rather the size of the file. Fortunately, this turns out to be trivially easy, thanks to the `os.path.getsize` function (this returns the same value that you would get from `os.stat`):

```
for currentdir, dirnames, filenames in os.walk('.'):
    for one_filename in filenames:
        first_part, ext = os.path.splitext(one_filename)
        try:
            counts[ext] +=
 ↪os.path.getsize(os.path.join(currentdir,one_filename))
        except FileNotFoundError:
            pass
```

The above code includes three changes from the previous version:

1. As indicated, this no longer adds 1 to the count for each extension, but rather the size of the file, which comes from `os.path.getsize`.

2. `os.path.join` puts the path and filename together and (as a bonus) uses the current operating system's path separation character. What are the odds of a program being used on a Windows system and, thus, needing a backslash rather than a slash? Pretty slim, but it doesn't hurt to use this sort of built-in operation.

3. `os.walk` doesn't normally look at symbolic links, which means you potentially can get yourself into some trouble trying to measure the sizes of files that don't exist. For this reason, here the counting is wrapped in a `try/except` block.

Once this is done, you can identify the file types consuming the greatest amount of space in the directory:

```
In [46]: for extension, count in counts.most_common(10):
    ...:     print(f"{extension:8}{count}")
    ...:
.pack   669153001
.zip    486110102
.ipynb  223155683
.sql    125443333
        46296632
.json   14224651
.txt    10921226
.pdf    7557943
.py     5253208
.pyc    4948851
```

Now things seem a bit different! In my case, it looks like I've got a lot of stuff in .pack files, indicating that my Git repository (where I store all of my old training examples, exercises and Jupyter notebooks) is quite large. I have a lot in zipfiles, in which I store my daily updates. And of course, lots in Jupyter notebooks, which are written in JSON format and can become quite large. The surprise to me is the .sql extension, which I honestly had forgotten that I had.

Files per Year

What if you want to know how many files of each type were modified in each year? This could be useful for removing logfiles or (if you're like me) identifying what large, unnecessary files are taking up space.

In order to do that, you'll need to get the modification time (`mtime`, in UNIX parlance) for each file. You'll then need to convert that `mtime` from a UNIX time (that is, the number of seconds since January 1st, 1970) to something you can parse and use.

Instead of using a `Counter` object to keep track of things, you can just use a dictionary. However, this dict's values will be a `Counter`, with the years serving as keys and the counts as values. Since you know that all of the main dicts will be `Counter` objects, you can just use a `defaultdict`, which will require you to write less code.

Here's how you can do all of this:

```
from collections import defaultdict, Counter
from datetime import datetime

counts = defaultdict(Counter)

for currentdir, dirnames, filenames in os.walk('.'):
    for one_filename in filenames:
        first_part, ext = os.path.splitext(one_filename)
        try:
            full_filename = os.path.join(currentdir,
 ↪one_filename)
            mtime =
 ↪datetime.fromtimestamp(os.path.getmtime(full_filename))
            counts[ext][mtime.year] += 1
        except FileNotFoundError:
            pass
```

First, this creates `counts` as an instance of `defaultdict` with a `Counter`. This means if you ask for a key that doesn't yet exist, the key will be created, with its value being a new `Counter` that allows you to say something like this:

```
counts['.zip'][2018] += 1
```

without having to initialize either the `zip` key (for counts) or the `2018` key (for the `Counter` object). You can just add one to the count and know that it's working.

Then, when you iterate over the filesystem, you grab the `mtime` from the filename (using `os.path.getmtime`). That is turned into a `datetime` object with `datetime.fromtimestamp`, a great function that lets you move from UNIX timestamps to human-style dates and times. Finally, you then add 1 to your counts.

Once again, you can display the results:

```
for extension, year_counts in counts.items():
    print(extension)
    for year, file_count in sorted(year_counts.items()):
        print(f"\t{year}\t{file_count}")
```

The `counts` variable is now a `defaultdict`, but that means it behaves just like a dictionary in most respects. So, you can iterate over its keys and values with `items`, which is shown here, getting each file extension and the `Counter` object for each.

Next the extension is printed, and then it iterates over the years and their counts, sorting them by year and printing them indented somewhat with a tab (`\t`) character. In this way, you can see precisely how many files of each extension have been modified per year—and perhaps understand which files are truly important and which you easily can get rid of.

Conclusion

Python can't and shouldn't replace Bash for simple scripting, but in many cases, if you're working with large number of files and/or creating reports, Python's standard library can make it easy to do such tasks with a minimum of code. ■

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

Have a Plan for Netplan

Ubuntu changed networking. Embrace the YAML.

By Shawn Powers

Shawn Powers is Associate Editor here at *Linux Journal*, and has been around Linux since the beginning. He has a passion for open source, and he loves to teach. He also drinks too much coffee, which often shows in his writing.

If I'm being completely honest, I still dislike the switch from `eth0, eth1, eth2` to names like, `enp3s0, enp4s0, enp5s0`. I've learned to accept it and mutter to myself while I type in unfamiliar interface names. Then I installed the new LTS version of Ubuntu and typed `vi /etc/network/interfaces`. Yikes. After a technological lifetime of entering my server's IP information in a simple text file, that's no longer how things are done. Sigh. The good news is that while figuring out Netplan for both desktop and server environments, I fixed a nagging DNS issue I've had for years (more on that later).

The Basics of Netplan

The old way of configuring Debian-based network interfaces was based on the `ifupdown` package. The new default is called Netplan, and although it's not terribly difficult to use, it's drastically different. Netplan is sort of the interface used to configure the back-end dæmons that actually configure the interfaces. Right now, the back ends supported are NetworkManager and `networkd`.

If you tell Netplan to use NetworkManager, all interface configuration control is handed off to the GUI interface on the desktop. The NetworkManager program itself hasn't changed;

it's the same GUI-based interface configuration system you've likely used for years.

If you tell Netplan to use `networkd`, systemd itself handles the interface configurations. Configuration is still done with Netplan files, but once "applied", Netplan creates the back-end configurations systemd requires. The Netplan files are vastly different from the old /etc/network/interfaces file, but it uses YAML syntax, and it's pretty easy to figure out.

The Desktop and DNS

If you install a GUI version of Ubuntu, Netplan is configured with NetworkManager as the back end by default. Your system should get IP information via DHCP or static entries you add via GUI. This is usually not an issue, but I've had a terrible time with my split-DNS setup and `systemd-resolved`. I'm sure there is a magical combination of configuration files that will make things work, but I've spent a lot of time, and it always behaves a little oddly. With my internal DNS server resolving domain names differently from external DNS servers (that is, split-DNS), I get random lookup failures. Sometimes `ping` will resolve, but `dig` will not. Sometimes the internal A record will resolve, but a `CNAME` will not. Sometimes I get resolution from an external DNS server (from the internet), even though I never configure anything other than the internal DNS!

I decided to disable `systemd-resolved`. That has the potential to break DNS lookups in a VPN, but I haven't had an issue with that. With `resolved` handling DNS information, the /etc/resolv.conf file points to 127.0.0.53 as the nameserver. Disabling `systemd-resolved` will stop the automatic creation of the file. Thankfully, NetworkManager itself can handle the creation and modification of /etc/resolv.conf. Once I make that change, I no longer have an issue with split-DNS resolution. It's a three-step process:

1. Do `sudo systemctl disable systemd-resolved.service`.

2. Then `sudo rm /etc/resolv.conf` (get rid of the symlink).

3. Edit the /etc/NetworkManager/NetworkManager.conf file, and in the `[main]` section, add a line that reads `DNS=default`.

Once those steps are complete, NetworkManager itself will create the /etc/resolv.conf file, and the DNS server supplied via DHCP or static entry will be used instead of a 127.0.0.53 entry. I'm not sure why the `resolved` dæmon incorrectly resolves internal addresses for me, but the above method has been foolproof, even when switching between networks with my laptop.

Netplan CLI Configuration

If Ubuntu is installed in server mode, it is almost certainly configured to use `networkd` as the back end. To check, have a look at the /etc/netplan/config.yaml file. The `renderer` should be set to `networkd` in order to use the `systemd-networkd` back end. The file should look something like this:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp2s0:
      dhcp4: true
```

Important note: remember that with YAML files, whitespace matters, so the indentation is important. It's also *very* important to remember that after making any changes, you need to run `sudo netplan apply` so the back-end configuration files are populated.

The default renderer is `networkd`, so it's possible you won't have that line in your configuration file. It's also possible your configuration file will be named something different in the /etc/netplan folder. All .conf files are read, so it doesn't matter what it's called as long as it ends with .conf. Static configurations are fairly simple to set up:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp2s0:
      dhcp4: no
      addresses:
        - 192.168.1.10/24
        - 10.10.10.10/16
      gateway4: 192.168.1.1
      nameservers:
        addresses: [192.168.1.1, 8.8.8.8]
```

Notice I've assigned multiple IP addresses to the interface. Netplan does not support virtual interfaces like `enp3s0:0`, rather multiple IP addresses can be assigned to a single interface.

Unfortunately, `networkd` doesn't create an /etc/resolv.conf file if you disable the `resolved` dæmon. If you have problems with split-DNS on a headless computer, the best solution I've come up with is to disable `systemd-resolved` and then manually create an /etc/resolv.conf file. Since headless computers don't usually move around as much as laptops, it's likely the /etc/resolv.conf file won't need to be changed. Still, I wish `networkd` had an option to manage the resolv.conf file the same way NetworkManager does.

Advanced Network Configurations

The configuration formats are different, but it's still possible to do more advanced network configurations with Netplan:

Bonding:

```
network:
  version: 2
  renderer: networkd
  bonds:
    bond0:
      dhcp4: yes
      interfaces:
        - enp2s0
        - enp3s0
      parameters:
        mode: active-backup
        primary: enp2s0
```

The various bonding modes (`balance-rr`, `active-backup`, `balance-xor`, `broadcast`, `802.3ad`, `balance-tlb` and `balance-alb`) are supported.

Bridging:

```
network:
  version: 2
  renderer: networkd
  bridges:
    br0:
      dhcp4: yes
      interfaces:
        - enp4s0
        - enp3s0
```

Bridging is even simpler to set up. This configuration creates a bridge device using the two interfaces listed. The device (`br0`) gets address information via DHCP.

CLI Networking Commands

If you're a crusty old sysadmin like me, you likely type `ifconfig` to see IP information without even thinking. Unfortunately, those tools are not usually installed by default. This isn't actually the fault of Ubuntu and Netplan; the old `ifconfig` toolset has been deprecated. If you want to use the old `ifconfig` tool, you can install the package:

```
sudo apt install net-tools
```

But, if you want to do it the "correct" way, the new "ip" tool is the proper way to do it. Here are some equivalents of things I commonly do with `ifconfig`:

Show network interface information.

Old way:

```
ifconfig
```

New way:

```
ip address show
```

(Or you can just do `ip a`, which is actually less typing than `ifconfig`.)

Bring interface up.

Old way:

```
ifconfig enp3s0 up
```

New way:

```
ip link set enp3s0 up
```

Assign IP address.

Old way:

```
ifconfig enp3s0 192.168.1.22
```

New way:

```
ip address add 192.168.1.22 dev enp3s0
```

Assign complete IP information.

Old way:

```
ifconfig enp3s0 192.168.1.22 net mask 255.255.255.0 broadcast
 ↪192.168.1.255
```

New way:

```
ip address add 192.168.1.22/24 broadcast 192.168.1.255
 ↪dev enp3s0
```

Add alias interface.

Old way:

```
ifconfig enp3s0:0 192.168.100.100/24
```

New way:

```
ip address add 192.168.100.100/24 dev enp3s0 label enp3s0:0
```

Show the routing table.

Old way:

```
route
```

New way:

```
ip route show
```

Add route.

Old way:

```
route add -net 192.168.55.0/24 dev enp4s0
```

New way:

```
ip route add 192.168.55.0/24 dev enp4s0
```

Old Dogs and New Tricks

I hated Netplan when I first installed Ubuntu 18.04. In fact, on the particular server I was installing, I actually started over and installed 16.04 because it was "comfortable". After a while, curiosity got the better of me, and I investigated the changes. I'm still more comfortable with the old /etc/network/interfaces file, but I have to admit, Netplan makes a little more sense. There is a single "front end" for configuring networks, and it uses different back ends for the heavy lifting. Right now, the only back ends are the GUI NetworkManager and the `systemd-networkd` dæmon. With the modular system, however, that could change someday without the need to learn a new way of configuring interfaces. A simple change to the `renderer` line would send the configuration information to a new back end.

With regard to the new command-line networking tool (`ip` vs. `ifconfig`), it really behaves more like other network devices (routers and so on), so that's probably a good change as well. As technologists, we need to be ready and eager to learn new things. If we weren't always trying the next best thing, we'd all be configuring Trumpet Winsock to dial in to the internet on our Windows 95 machines. I'm glad I tried that new Linux thing, and while it wasn't quite as dramatic, I'm glad I tried Netplan as well! ■

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

Normalizing Filenames and Data with Bash

URLify: convert letter sequences into safe URLs with hex equivalents.

By Dave Taylor

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

This is my 155th column. That means I've been writing for *Linux Journal* for:

```
$ echo "155/12" | bc
12
```

No, wait, that's not right. Let's try that again:

```
$ echo "scale=2;155/12" | bc
12.91
```

Yeah, that many years. Almost 13 years of writing about shell scripts and lightweight programming within the Linux environment. I've covered a lot of ground, but I want to go back to something that's fairly basic and talk about filenames and the web.

It used to be that if you had filenames that had spaces in them, bad things would happen: "my mom's cookies.html" was a recipe for disaster, not good cookies—um, and not those sorts of web cookies either!

As the web evolved, however, encoding of special characters became the norm, and every Web browser had to be able to manage it, for better or worse. So spaces became either "+" or %20 sequences, and everything else that wasn't a regular alphanumeric character was replaced by its hex ASCII equivalent.

In other words, "my mom's cookies.html" turned into "my+mom%27s+cookies.html" or "my%20mom%27s%20cookies.html". Many symbols took on a second life too, so "&" and "=" and "?" all got their own meanings, which meant that they needed to be protected if they were part of an original filename too. And what about if you had a "%" in your original filename? Ah yes, the recursive nature of encoding things....

So purely as an exercise in scripting, let's write a script that converts any string you hand it into a "web-safe" sequence. Before starting, however, pull out a piece of paper and jot down how you'd solve it.

Normalizing Filenames for the Web

My strategy is going to be easy: pull the string apart into individual characters, analyze each character to identify if it's an alphanumeric, and if it's not, convert it into its hexadecimal ASCII equivalent, prefacing it with a "%" as needed.

There are a number of ways to break a string into its individual letters, but let's use Bash string variable manipulations, recalling that `${#var}` returns the number of characters in variable `$var`, and that `${var:x:1}` will return just the letter in `$var` at position `x`. Quick now, does indexing start at zero or one?

Here's my initial loop to break `$original` into its component letters:

```
input="$*"

echo $input
```

```
for (( counter=0 ; counter < ${#input} ; counter++ ))
do
   echo "counter = $counter -- ${input:$counter:1}"
done
```

Recall that `$*` is a shortcut for everything from the invoking command line other than the command name itself—a lazy way to let users quote the argument or not. It doesn't address special characters, but that's what quotes are for, right?

Let's give this fragmentary script a whirl with some input from the command line:

```
$ sh normalize.sh "li nux?"
li nux?
counter = 0 -- l
counter = 1 -- i
counter = 2 --
counter = 3 -- n
counter = 4 -- u
counter = 5 -- x
counter = 6 -- ?
```

There's obviously some debugging code in the script, but it's generally a good idea to leave that in until you're sure it's working as expected.

Now it's time to differentiate between characters that are acceptable within a URL and those that are not. Turning a character into a hex sequence is a bit tricky, so I'm using a sequence of fairly obscure commands. Let's start with just the command line:

```
$ echo '~' | xxd -ps -c1 | head -1
7e
```

Now, the question is whether "~" is actually the hex ASCII sequence 7e or not.

A quick glance at http://www.asciitable.com confirms that, yes, 7e is indeed the ASCII for the tilde. Preface that with a percentage sign, and the tough job of conversion is managed.

But, how do you know what characters can be used as they are? Because of the weird way the ASCII table is organized, that's going to be three ranges: 0–9 is in one area of the table, then A–Z in a second area and a–z in a third. There's no way around it, that's three range tests.

There's a really cool way to do that in Bash too:

```
if [[ "$char" =~ [a-z] ]]
```

What's happening here is that this is actually a regular expression (the `=~`) and a range `[a-z]` as the test. Since the action I want to take after each test is identical, it's easy now to implement all three tests:

```
if [[ "$char" =~ [a-z] ]]; then
  output="$output$char"
elif [[ "$char" =~ [A-Z] ]]; then
  output="$output$char"
elif [[ "$char" =~ [0-9] ]]; then
  output="$output$char"
else
```

As is obvious, the `$output` string variable will be built up to have the desired value.

What's left? The hex output for anything that's not an otherwise acceptable character. And you've already seen how that can be implemented:

```
hexchar="$(echo "$char" | xxd -ps -c1 | head -1)"
 output="$output%$hexchar"
```

A quick run through:

```
$ sh normalize.sh "li nux?"
li nux? translates to li%20nux%3F
```

See the problem? Without converting the hex into uppercase, it's a bit weird looking. What's "nux"? That's just another step in the subshell invocation:

```
hexchar="$(echo "$char" | xxd -ps -c1 | head -1 | \
   tr '[a-z]' '[A-Z]')"
```

And now, with that tweak, the output looks good:

```
$ sh normalize.sh "li nux?"
li nux? translates to li%20nux%3F
```

What about a non-Latin-1 character like an umlaut or an n-tilde? Let's see what happens:

```
$ sh normalize.sh "Señor Günter"
Señor Günter translates to Se%C3B1or%200AG%C3BCnter
```

Ah, there's a bug in the script when it comes to these two-byte character sequences, because each special letter should have two hex byte sequences. In other words, it should be converted to `se%C3%B1or g%C3%BCnter` (I restored the space to make it a bit easier to see what I'm talking about).

In other words, this gets the right sequences, but it's missing a percentage sign—`%C3B` should be `%C3%B`, and `%C3BC` should be `%C3%BC`.

Undoubtedly, the problem is in the `hexchar` assignment subshell statement:

```
hexchar="$(echo "$char" | xxd -ps -c1 | head -1 | \
    tr '[a-z]' '[A-Z]')"
```

Is it the `-c1` argument to `xxd`? Maybe. I'm going to leave identifying and fixing the problem as an exercise for you, dear reader. And while you're fixing up the script to support two-byte characters, why not replace "%20" with "+" too?

Finally, to make this maximally useful, don't forget that there are a number of symbols that are valid and don't need to be converted within URLs too, notably the set of "-_./!@#=&?", so you'll want to ensure that they don't get hexified (is that a word?). ■

Send comments or feedback via http://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.

Dealing with printk()

It's odd that **printk()** would pose so many problems for kernel development, given that it's essentially just a replacement for **printf()** that doesn't require linking the standard **C library** into the kernel.

And yet, it's famously a mess, full of edge cases, corner cases, deadlocks, race conditions and a variety of other tough-to-solve problems. The reason for this is, unlike printf(), the printk() system call has to produce reasonable behavior even when the entire system is in the midst of crashing. That's really the whole point—printk() needs to report errors and warnings that can be used to debug whatever strange and unexpected catastrophe has just hit a running system.

Trying to fix all the deadlocks and other problems at the same time would be too large a task for anyone, especially since each one is a special case defined by the particular context in which the printk() call appeared. But, sometimes a bunch of instances in a particular region of code can be addressed all together.

Sergey Senozhatsky recently tried to address some printk() deadlocks, although he acknowledged he wouldn't address any instances that were caused by the printk() code itself triggering a separate recursive printk() call. He wanted to concern himself with non-recursion-based deadlocks only.

Sergey focused on the console code, which was where printk() generally sent its output, and which was one place where printk() could deadlock. He added a very small safeguard to the code, but the result seemed to be that drivers all throughout the kernel would have to be updated to use the new safeguard.

His code was not met with universal acclaim. **Alan Cox** noticed that Sergey's safeguard added code to the "fast path"—a region of code that needed to be as fast and efficient as possible, because it was run all the time, many times per second. Slowing down the fast path would slow down the whole system. Alan suggested instead of this, it would be better for the kernel simply not to call printk() if the console code would be in a position to deadlock.

Sergey was not in any way satisfied, however. He pointed out that his patch solved real-world problems that users had reported experiencing directly. He didn't see how it would help anything simply to pull out the printk() instances that triggered the problem, especially if those instances were doing important work like reporting on the real reason the system was crashing and so on.

Sergey wanted to keep the printk() instances and implement the safeguards to protect them. However, at this point **Linus Torvalds** joined the discussion, saying:

> The rule is simple: DO NOT DO THAT THEN.
>
> Don't make recursive locks. Don't make random complexity. Just stop doing the thing that hurts.
>
> There is no valid reason why an UART driver should do a printk() of any sort inside the critical region where the console is locked.

> Just remove those printks, don't add new crazy locking.
>
> If you had a spinlock that deadlocked because it was inside an already spinlocked region, you'd say "that's buggy".
>
> This is the exact same issue. We don't work around buggy garbage. We fix the bug—by removing the problematic printk.

Sergey pointed out that the printk() instances were called from all those drivers he wanted to change. It wasn't a case of some simple part of the kernel having an extra printk(). The drivers all needed to be updated with the safeguard, or they would continue to report the wrong thing.

The conversation ended with no conclusion. It's difficult to know when something should be fixed versus removed. There are all sorts of technical questions that come up, including wondering if the fix is worth all the fuss.

Internationalizing the Kernel

At a time when many companies are rushing to internationalize their products and services to appeal to the broadest possible market, the Linux kernel is actively resisting that trend, although it already has taken over the broadest possible market—the infrastructure of the entire world.

David Howells recently created some sample code for a new kernel library, with some complex English-language error messages that were generated from several sources within the code. **Pavel Machek** objected that it would be difficult to automate any sort of translations for those messages, and that it would be preferable simply to output an error code and let something in userspace interpret the error at its leisure and translate it if needed.

In this case, however, the possible number of errors was truly vast, based on a variety of possible variables. David argued that representing each and every one with a single error code would use a prohibitively large number of error codes.

Ordinarily, I might expect Pavel to be on the winning side of this debate, with Linus Torvalds or some other top developer insisting that support for internationalization was necessary in order to give the best and most useful possible experience to all users.

However, Linus had a very different take on the situation:

> We don't internationalize kernel strings. We never have. Yes, some people tried to do some database of kernel messages for translation purposes, but I absolutely refused to make that part of the development process. It's a pain.
>
> For some GUI project, internationalization might be a big deal, and it might be "TheRule(tm)". For the kernel, not so much. We care about the technology, not the language.
>
> So we'll continue to give error numbers for "an error happened". And if/when people need more information about just what _triggered_ that error, they are as English-language strings. You can quote them and google them without having to understand them. That's just how things work.
>
> [...]
>
> There are places where localization is a good idea. The kernel is *not* one of those places.

He added later:

> I really think the best option is "Ignore the problem". The system calls will still continue to report the basic error numbers (EINVAL etc), and the extended error strings will be just that: extended error strings. Ignore them if you can't understand them.
>
> That said, people have wanted these kinds of extended error descriptors forever, and the reason we haven't added them is that it generally is more pain than it is necessarily worth.

Pavel still felt that, since David's code was all new, there was no ancient cruft standing in the way of implementing internationalization in this one new area. He agreed there was no point in a lot of other cases, but for this one, it felt like being given a fresh chance.

But Linus said, "Really. No translation. No design for translation. It's a nasty nasty rat-hole, and it's a pain for everybody."

He added, "the fact is, I want simple English interfaces. And people who have issues with that should just not use them. End of story. Use the existing error numbers if you want internationalization, and live with the fact that you only get the very limited error number. It's really that simple."

The discussion ended shortly thereafter. It's a fascinating rejection of a very politically popular attitude, based on the technical consideration that keeping the programming interface simple is worth more than keeping the user interface friendly.

Keeping Control in the Hands of the User

Various efforts always are underway to implement **Secure Boot** and to add features that will allow vendors to lock users out of controlling their own systems. In that scenario, users would look helplessly on while their systems refused to boot any kernels but those controlled by the vendors.

The vendors' motivation is clear—if they control the kernel, they can then stream media on that computer without risking copyright infringement by the user. If the vendor doesn't control the system, the user might always have some secret piece of software ready to catch and store any streamed media that could then be shared with others who would not pay the media company for the privilege.

Recently, **Chen Yu** and other developers tried to submit patches to enhance Secure Boot so that when the user hibernated the system, the kernel itself would encrypt its running image. This would appear to be completely unnecessary, since as Pavel Machek pointed out, there is already **uswsusp** (userspace software suspend), which encrypts the running image before suspending the system. As Pavel said, the only difference was that uswusp ran in userspace and not kernel space.

Perhaps in an effort to draw Chen into admitting the deeper motives behind the patch submission, Pavel asked Chen to elucidate exactly what security hole his patches addressed and how they would deal with them. Pavel would ask that question over and over again before the end of the discussion, and he would not receive an answer.

Chen offered a variety of justifications for the patch, including letting users do less work, but none of them answered the fundamental question: why was this patch needed as a security enhancement in the first place? And eventually, Pavel called it like he saw it. He said, "Purpose here is to prevent the user from reading/modifying kernel memory content on machine he owns. Strange as it may sound, that is what 'secure' boot requires (and what **Disney** wants)."

The discussion ended inconclusively, but not utterly. It's clear that Pavel, and a group of core kernel developers including Linus Torvalds, will continue to guard against allowing vendors to control user systems. This seems to be one of the fundamental values of the Linux kernel—to prevent the reemergence of the kind of situation we had in the 1980s, where vendors had ultimate control over virtually all software, while users were at the mercy of business decisions they didn't agree with but could do nothing about.

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 http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

DEEP DIVE

PROGRAMMING

Understanding Bash: Elements of Programming

Ever wondered why programming in Bash is so difficult? Bash employs the same constructs as traditional programming languages; however, under the hood, the logic is rather different.

By Vladimir Likic

The Bourne-Again SHell (Bash) was developed by the Free Software Foundation (FSF) under the GNU Project, which gives it a somewhat special reputation within the Open Source community. Today, Bash is the default user shell on most Linux installations. Although Bash is just one of several well known UNIX shells, its wide distribution with Linux makes it an important tool to know.

The main purpose of a UNIX shell is to allow users to interact effectively with the system through the command line. A common shell action is to invoke an executable, which in turn causes the kernel to create a new running process. Shells have mechanisms to send the output of one program as input into another and facilities to interact with the filesystem. For example, a user can traverse the filesystem or direct the output of a program to a file.

Although Bash is primarily a command interpreter, it's also a programming language.

Bash supports variables, functions and has control flow constructs, such as conditional statements and loops. However, all of this comes with some unusual quirks. This is because Bash attempts to fulfill two roles at the same time: to be a command interpreter and a programming language—and there is tension between the two.

All UNIX shells, including Bash, are primarily command interpreters. This trait has a deep history, stretching all the way to the very first shell and the first UNIX system. Over time, UNIX shells acquired the programming capabilities by evolution, and this has led to some unusual solutions for the programming environment. As many people come to Bash already having some background in traditional programming languages, the unusual perspective that Bash takes with programming constructs is a source of much confusion, as evidenced by many questions posted on Bash forums.

In this article, I discuss how programming constructs in Bash differ from traditional programming languages. For a true understanding of Bash, it's useful to understand how UNIX shells evolved, so I first review the relevant history, and then introduce several Bash features. The majority of this article shows how the unusual aspects of Bash programming originate from the need to blend the command interpreter function seamlessly with the capabilities of a programming language.

Doing Two Different Things at Once

The original Thompson shell was a simple command interpreter whose mode of operation was as follows:

```
$ command [ arg1 ... [ argN ]
```

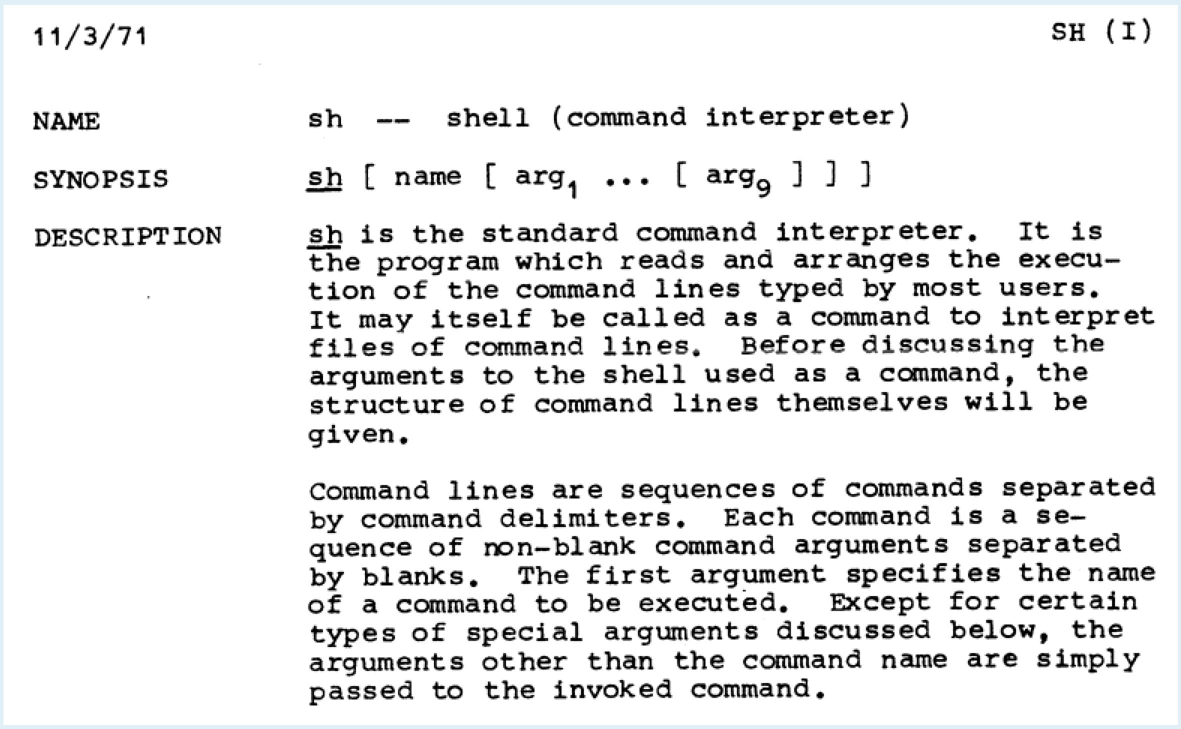
where `command` is the name of the executable file (that is, a command to be executed), and the optional arguments `arg1 ... argN` are passed to the command. The Thompson shell had no programming capabilities. This changed with the development of the Mashey shell (and later the Bourne shell). In his seminal paper "The UNIX Shell", published in 1978, Stephen Bourne wrote:

> The UNIX shell is both a programming language and a command language. As a

Bash History

The term "shell" originated from the MULTICS project, a collaboration between Massachusetts Institute of Technology (MIT), General Electric and Bell Telephone Laboratories (henceforth Bell Labs) to develop a next-generation time-sharing operating system. Unhappy with the progress, Bell Labs withdrew from the project in 1969, and the Bell Labs team who worked on MULTICS went on to develop their own operating system: UNIX.

The ancestor of Bash is the Thompson shell, the first UNIX command interpreter, developed by Ken Thompson in 1971. Figure 1 shows an



```
11/3/71                                                SH (I)

NAME            sh  --  shell (command interpreter)

SYNOPSIS        sh [ name [ arg1 ... [ arg9 ] ] ]

DESCRIPTION     sh is the standard command interpreter.  It is
                the program which reads and arranges the execu-
                tion of the command lines typed by most users.
                It may itself be called as a command to interpret
                files of command lines.  Before discussing the
                arguments to the shell used as a command, the
                structure of command lines themselves will be
                given.

                Command lines are sequences of commands separated
                by command delimiters.  Each command is a se-
                quence of non-blank command arguments separated
                by blanks.  The first argument specifies the name
                of a command to be executed.  Except for certain
                types of special arguments discussed below, the
                arguments other than the command name are simply
                passed to the invoked command.
```

Figure 1. An Excerpt from the *UNIX Programming Manual*, 1st Edition, Published in 1971, Describing the Original Thompson Shell

excerpt from the *UNIX Programming Manual*, 1st edition, that describes the Thompson shell.

Between 1973–1975, John R. Mashey extended the original Thompson shell and added several programming capabilities, making it a high-level programming language. In Mashey's own words:

> Modifications have been aimed at improving the use of the shell...and making it even more convenient to use as a high-level programming language. In line with the philosophy of much existing UNIX software, an attempt has been made to add new features only when they are shown necessary by actual user experience in order to avoid contaminating a compact, elegant system through "creeping featurism". (From J. Mashey, "Using a Command Language as a High-level Programming Language", CSE '76 Proceedings of the 2nd International Conference on Software engineering, 1976.)

Stephen Bourne started working on a new shell early in 1976. The Bourne shell benefited from the concepts introduced by the Mashey shell, and it brought some new ideas of its own. The Bourne shell officially was introduced in UNIX Version 7, released in 1979.

The original Thompson shell, the Mashey shell and the Bourne shell were all called sh, and they overlapped or replaced one another in the years 1970–1976 as they were refined and gained additional capabilities. Throughout 1970s, UNIX was mostly being developed at Bell Labs and, in parallel, at the University of California at Berkeley (the variant known as BSD). With the development of UNIX, shells were constantly developed and refined. At the time when the Bourne shell already was in use, Bill Joy at Berkeley developed the C shell (csh). The C shell was the first truly alternative UNIX shell, and it was incorporated in the 2BSD release of Berkeley UNIX. In the early 1980s, David Korn developed the Korn shell (ksh). Compared to the Bourne shell, the C shell emphasized the command interpreter mode, and the Korn shell came with more extensive programming capabilities.

UNIX development efforts at Bell Labs and Berkeley enriched each other, and the two versions were later merged. In the 1980s, AT&T licensed UNIX to a number of commercial vendors, and this resulted in the disruptive wars for the UNIX market domination. In 1985, Richard Stallman established the Free Software Foundation (FSF), whose main initiative was to build a free-to-use UNIX-like system, one that is not encumbered by the intellectual property issues surrounding UNIX. This is the famous GNU Project ("GNU's not UNIX"). In fact, the original letter from Stallman, sent on the net.unix-wizards mailing list in September 1983, started with the cry: "Free Unix!"

Since it's impossible to have free UNIX without a shell, that was a priority for the GNU Project. Brian Fox, the Free Software Foundation's first paid programmer, started working on a shell 1988. This became Bash, first released as beta in 1989. Bash is mostly a clone of the Bourne shell (hence "Bourne-Again"), but it also includes additional features inspired by the C shell and Korn shell. Brian Fox was the official maintainer of Bash until 1992. At the time, Chet Ramey already was involved with the work on Bash, and he became the official maintainer in 1993. Chet Ramey continued to maintain and develop Bash for the next 25 years, and he's still Bash's current maintainer.

> programming language, it contains control-flow primitives and string-valued variables. As a command language, it provides a user interface to the process-related facilities of the UNIX operating system. (S.R Bourne, "The UNIX Shell", *The Bell System Technical Journal*, Vol 56, No 6, July–August 1978.)

Note the emphasis on the different functionality: a programming language and a command language. In fact, it was the Mashey and Bourne shells that extended the capabilities of the Thompson shell beyond the command interpreter. The shell's original role was a command interpreter, and the programming capabilities of shells were added later. UNIX shells evolved some ingenious ways of consolidating the programming capabilities with the original command interpreter role.

Bash Mode of Operation

Today's Bash is more powerful compared to the original Mashey shell and the Bourne shell. However, the purpose of the shell remains exactly the same. Arguably, the most important function of the shell is running commands (that is, submitting an executable file to the kernel for execution). This has several profound ramifications. For a start, Bash treats (almost) anything that is given to it as a command. Consider the following Bash session:

```
$ VAR
bash: VAR: command not found
$ 9
bash: 9: command not found
$ 9 + 1
bash: 9: command not found
$
```

This shows that Bash splits the input into words, then attempts to execute the first word as a command (the "words" `VAR` and `9`). Here, a "command" may be either a Bash built-in command (such as `cd`), a utility (such as `/bin/ls`) or some other executable file. When the string `9 + 1` was given on input, Bash split it into three "words": `9`, `+` and `1`. It's important to note that Bash keeps all words as strings and has no concept of numbers until forced to do an arithmetic evaluation. As a rather simplified summary, Bash operates as follows:

1. Takes the input and splits it into words on white spaces (space or tab).

2. Assumes that the first word is a command. If anything follows the first word, it assumes they are arguments to be passed to the command.

3. Attempts to execute the command (and pass the arguments to it, if any).

This view ignores several intermediate steps. For example, Bash scans the input line and performs all sorts of expansions and replacements. It also checks for a built-in command with the name given, and executes that, if it exists. Not to lose sight of the big picture, I often ignore these details.

So, Bash's most essential purpose is to execute commands, and this has some profound implications. Notably, the programming constructs in Bash, which at first sight may look like a programming language, are derived from this mode of operation. And, that is the central theme of this article.

Bash Built-ins vs. External Commands

The point that's often confusing to Bash newcomers is the difference between Bash built-in commands and external commands. On a typical Linux/UNIX system, a number of common commands are both built-in in Bash and *also* exist as independent executables with the same name. Examples of this include `echo` (built-in) and `/bin/echo`, `kill` (built-in) and `/bin/kill`, `test` (built-in) and `/usr/bin/test` (and there are more). Consider how the Bash built-in `echo` and `/bin/echo` behave very similarly:

```
$ echo 'Echoed with a built-in!'
Echoed with a built-in!
$ /bin/echo 'Echoed with external program!'
Echoed with external program!
$
```

However, there are also subtle differences (try `echo --version`). Why this duplication of commands? There are several reasons. The built-in version typically exists for performance reasons: Bash built-ins execute within the shell process that's already running. In contrast, executing an external utility involves loading and executing the external binary by the kernel, which is a much slower process.

At this point, it's useful to note that some shell commands, by their nature, cannot be external utilities (in other words, they must be shell built-ins). Consider the `cd` command that changes the current working directory. An external utility wouldn't be able to change the shell's current working directory, so `cd` must be a Bash built-in. Why? Because invoking a command as an external utility would make the shell its parent process, and a child process cannot change the current working directory of the parent process.

You could turn this question around and ask, "if `echo` is already built in to the shell, why does the external utility `/bin/echo` exist?" That's because one doesn't always work through the shell and may need to invoke `echo` without the mediating shell process. Second, in principle, there's nothing to enforce that a UNIX shell must have `echo` as a built-in, and therefore, it's important to have the external utility `/bin/echo` as a fallback.

A practical problem users often face is this: how do you know whether the command you just called is the shell built-in or an external utility with the same name? The Bash command `type` (which is itself a shell built-in) indicates what command would be used if executed. For example:

```
$ type echo
echo is a shell builtin
$ type ls
ls is hashed (/bin/ls)
$
```

The basic rule is as follows: if the built-in command with a given name exists, it will be executed. If the built-in command doesn't exist, Bash will search for an external program, and if found, will execute it. If you want to be sure to use the executable, which happens to have the same name as a shell built-in, calling the executable with the full path will do.

Variable Assignment

When a command is entered in Bash, Bash expects that the first word it encounters is a command. However, there's one exception: if the first word contains `=`, Bash will attempt to execute a variable assignment. For example:

```
$ VAR=7
$
```

This has assigned the value 7 to the variable named `VAR`. To retrieve the value of a variable, you need to prefix the variable name with the dollar sign. Thus, to view the value of a variable, you can combine the dollar-sign prefix with `echo`:

```
$ echo $VAR
7
$
```

For a variable assignment, a contiguous string that contains `=` is important. The following will fail:

```
$ VAR = 1
bash: VAR: command not found
$
```

In this case, Bash splits the input `VAR = 1` into three "words" (`VAR`, `=` and `1`) and then attempts to execute the first word as a command. This clearly isn't what was intended here.

The ? Built-in Variable

Although Bash allows you to create arbitrary variables on the fly simply by assigning the values to them, it also has a number of built-in variables. An example of a built-in variable is `BASHPID`. This contains the process ID of the Bash shell itself:

```
$ echo $BASHPID
2141
$
```

Another built-in variable (and one that I cover extensively here) is `?`. At any point in a Bash session, this variable contains the return value of the last executed command. The return value is always an integer. (And specifically, this is the return value of the C program function `main()`. Note: in any C program the function `main()` must return an integer.) By the UNIX convention, the return value of 0 denotes success, and any other value denotes failure. For example, consider the utility `/bin/ls`:

```
$ touch NEWFILE
$ /bin/ls NEWFILE
NEWFILE
$ echo $?
0
$
```

As per the convention, the utility `/bin/ls` returned 0 on success, which you can see by inspecting the value of `?`. If `ls` is unable to execute (for example, unable to access the file), it returns the value `>0`:

```
$ /bin/ls DOESNOTEXIST
ls: cannot access 'DOESNOTEXIST': No such file or directory
$ echo $?
2
$ echo $?
0
$
```

In the last example, note that the first `?` was set to 2, and the second `?` was set to 0. Why? Because the second `?` contains the exit status of the `echo` command (which executed successfully). Remember, the `?` variable contains the exit status of the last executed command. You can use the commands `true` and `false` to set the value of `?` to 0 or 1, respectively:

```
$ false
$ echo $?
1
$ false
$ true
$ echo $?
0
$
```

That might look rather silly at first, but keep reading.

Bash Blending Behavior

Now let's consider how Bash provides an impression of a seamlessly integrated command environment, even when the tasks it executes are inherently quite different. First, note that running a Bash built-in command produces the same effect on the `?` variable as running an external program:

```
$ false   #  set ? to 1
$ echo 'Calling a built-in command'
Calling a built-in command
$ echo $?
0
```

This example shows that calling the built-in command `echo` changed `?` to 0 (to confirm this, first run the `false` command, which sets `?` to 1). The point is that it behaves the same as calling the external program `echo`:

```
$ false  #  set ? to 1
$ /bin/echo 'Calling external program'
Calling external program
$ echo $?
0
```

Yet, these two scenarios are quite different. In the first scenario, Bash invoked an internal command `echo`; in the second example, Bash requested from the kernel to run an external executable (`/bin/echo`) and suspended itself waiting for the executable to complete. The effect on the `?` variable is exactly the same.

Even for a variable assignment, Bash will set the `?` variable accordingly:

```
$ false   #  set ? to 1
$ VAR=one
$ echo $?
0
$
```

From this, you can see that Bash treats a variable assignment as a command. If the variable assignment is not successful, `?` is set to a value >0. For example, the built-in variable `BASHPID` is read-only, and you can't change it (that is, Bash can't change its own process ID). So this will fail:

```
$ true  #  set ? to 0
$ BASHPID=99
$ echo $?
1
$
```

Attempting to execute a non-existent command would also set `?` to indicate a failure:

```
$ true  # set ? to 0
$ DUMMY
bash: DUMMY: command not found
$ echo $?
127
$
```

In this case, Bash filled the special variable `?` with the number 127. This number is hard-wired in Bash, and it specifically means "command not found".

To summarize, the above examples show three completely different scenarios: invoking an internal Bash command, running an external program and variable assignment. Yet, Bash views all three as command execution and provides a common behavior with respect to the `?` special variable. Armed with these insights, now let's examine three basic programming constructs in Bash: the `if` statement, the `while` loop and the `until` loop.

The Conditional `if` Statement

The fundamental element of almost every programming language is the conditional `if` statement. In the C language, it looks like this:

```
if (TRUTH_TEST) {
   statements to execute
}
```

Here `TRUTH_TEST` is a test that evaluates true or false according to the rules of the C language. This is sometimes called "truth value testing". Here's an example of this in Python:

```
if True:
    print('Yay true!')
```

In Bash, the same example looks like this:

```
if true
then
  echo 'Yay true!'
fi
```

You can reformat this by using ; to provide a handy one-liner to type in:

```
$ if true; then echo 'Yay true!'; fi
Yay true!
$
```

This looks very much like the `if` conditional statement in any programming language. However, it's not. In the above example, `true` is a command. In fact, `true` is a shell built-in:

```
$ type true
true is a shell builtin
$ help true
true: true
    Return a successful result.

    Exit Status:
    Always succeeds.
```

Let that sink in: `true` is a command. In fact, that's the same `true` command that was run above from the command line to set the value of the `?` variable. What then is the `if` statement evaluating? It's evaluating the return value of the `true` command. If you're not convinced, consider that `true` can be replaced with the external utility `/bin/true`:

```
$ if /bin/true; then echo 'Yay true!'; fi
Yay true!
$
```

Where:

```
$ man true
TRUE(1)                    User Commands                   TRUE(1)

NAME
        true - do nothing, successfully

SYNOPSIS
        true [ignored command line arguments]
        true OPTION

DESCRIPTION
        Exit with a status code indicating success.
```

If `true` is a command, you can put any command there, right? Indeed:

```
$ if /bin/echo; then echo 'Yay true!'; fi

Yay true!
$
```

Notice how the blank line was printed before the string `Yay true!`. That's because the `if` statement actually executed the command `/bin/echo`, and without any arguments, this prints a newline character. You actually can give an argument to the `echo` command:

```
$ if /bin/echo 'Hi'; then echo 'Yay true!'; fi
Hi
Yay true!
$
```

The two `echo` commands executed here are different: the first is the external utility `/bin/echo`; the second, an `echo` that appears in the body of the `if` statement, is the shell built-in. Clearly, the second `echo` could be replaced with the external utility too.

Moving on, I mentioned previously that Bash will treat the variable assignment as a command. Thus, variable assignment can be used in the same place as the built-in command or external executable:

```
$ if VAR=99; then echo 'Assignment done!'; fi
Assignment done!
$ echo $VAR
99
$
```

To sum up, the general form of the `if` conditional statement is: `if CMD1; then CMD2; fi` where `CDM1` and `CMD2` are commands. The `if` statement controls flow by evaluating the exit code of the command `CMD1`: if `CMD1` was successful (judging by the exit status of 0), then `CMD2` is executed. This is rather different compared to truth value testing in most traditional programming languages, and it's the source of much confusion. I shall call this *source of confusion number 1*.

The `false` Command

I just described how `true` is a command. So not surprisingly, there is a `false`, the exact opposite of `true`. For the Bash built-in:

```
$ type false
false is a shell builtin
$ help false
false: false
    Return an unsuccessful result.

    Exit Status:
    Always fails.
$
```

And, there is an external utility with the same function:

```
$ man false
FALSE(1)                 User Commands                FALSE(1)

NAME
       false - do nothing, unsuccessfully

SYNOPSIS
       false [ignored command line arguments]
       false OPTION

DESCRIPTION
       Exit with a status code indicating failure.
```

The commands `true` and `false` do nothing, but exit with the status 0 or 1, respectively. Since the `if` statement evaluates the exit code when deciding whether to execute the body, `if true` always succeeds, and `if false` always fails. Note that the exit value of `true` is 0, and the exit value of `false` is 1. This is somewhat counterintuitive, and it's the exact opposite of most programming languages. For example, in Python truth value testing, 0 is equated with False (boolean), and 1 is equated with True (boolean).

Bash `if` Is Testing the Exit Value

Let's confirm that the `if` statement in Bash is merely testing the value of the program's exit value by writing a simple C program, true.c, that returns 1 (note, the real utility `true` returns 0, or success!):

```
int main() {
   return 1;
}
```

This program doesn't do much; it merely returns 1 as the exit status. According to the UNIX convention, the exit status of 1 indicates a failure (no matter that the program may run just fine!). Let's compile and execute this program, and confirm that it returns an "unsuccessful" exit status to the shell:

```
$ gcc true.c -o true
$ ./true
$ echo $?
1
$
```

So, if you use this program in the `if` statement, the output won't be what you may expect:

```
$ if ./true; then echo 'Yay true!'; fi
$
```

In other words, the `true` command has "failed". This example confirms that all the `if` statement does is evaluate the exit status. It doesn't matter that the program runs just fine, exactly as intended; from the Bash perspective, a non-zero exit status indicates failure. This I shall call the *source of confusion number 2*.

More Bash Ingenuity

Consider the following task: test if the file exists, and if it does, delete it. For this you can use the Bash built-in `test` command with the `-e` flag:

```
$ rm dum.txt        # make sure file 'dum.txt' doesn't exist
$ test -e dum.txt   # test if file 'dum.txt' exists
$ echo $?           # confirm that the command test failed
1
$ touch dum.txt     # now create file 'dum.txt'
$ test -e dum.txt   # test if file 'dum.txt' exists
$ echo $?           # confirm the command test was successful
0
$
```

Therefore, to test if the file exists, and if yes, delete it:

```
$ touch dum.txt    # create file 'dum.txt'
$ if test -e dum.txt; then rm dum.txt; fi  # file deleted
$
```

The key to note here is that `if test -e dum.txt; then rm dum.txt; fi` actually executes the command `test -e dum.txt`. In this case, `test` is a Bash built-in. As you might suspect, there is a `/usr/bin/test` utility that does the same thing and could be used to the same effect:

```
$ touch dum.txt  # create file 'dum.txt'
$ if /usr/bin/test -e dum.txt; then rm dum.txt; fi
 ↪# file deleted
$
```

Now, Bash implements `[ ]` as a synonym for the built-in `test` command:

```
$ test -e dum.txt  #  command successful if file exists
$ [ -e dum.txt ]   #  exactly the same as previous example!
```

Note, `[ -e dum.txt ]` is a command. And this, of course, returns 0 on success and 1 on failure. Let's confirm:

```
$ rm dum.txt
$ [ -e dum.txt ]
$ echo $?
1
$ touch dum.txt
$ [ -e dum.txt ]
$ echo $?
0
$
```

With this understanding, you can repeat the above example with the `[ -e ... ]` construct:

```
$ touch dum.txt  # create file 'dum.txt'
$ if [ -e dum.txt ]; then rm dum.txt; fi  # file deleted
$
```

The last construct looks even more like the `if` control statement in most traditional programming languages. However, it's not. `[ ]` is a command—basically another way to call the built-in `test` command.

Command Lists

The surprises don't quite end there. In Bash, the `if` statement can take any number of commands separated by a semicolon, after the keyword `if` and before the body denoted with the keyword `then`. Something like this: `if CMD1; CMD2; ... CMDN; then CMDN+1; CMDN+2; CMDN+M; fi`. The `if` statement evaluates all commands sequentially and executes the body of the loop only if the exit status of the last command is 0 (a success by convention). Consider the following example:

```
$ if false; true; then echo 'Yay true!'; fi
 ↪# body will execute
Yay true!
$ if true; false; then echo 'Yay true!'; fi
 ↪# body will not execute
$
```

So in Bash, it's completely legal to write something like this:

```
$ if [ -e dum.txt ]; echo 'Hi'; false; then rm dum.txt; fi
Hi
$
```

This executes three commands given after `if`, and it never will execute the body (`rm dum.txt`) because the last command is `false`, which always fails (more precisely, returns a non-zero status). In summary, in place of a single command, you can use a list of commands. The overall exit status of such a command list is given by the exit status of the last command in the list. This I shall call the *source of confusion number 3*.

The Loops `while` and `until`

Understanding the behavior of the `if` statement is rather useful because the same behavior applies to `while` and `until` loops. Consider the following example:

```
$ while true; do echo 'Hi, while looping ...'; done
Hi, while looping ...
Hi, while looping ...
Hi, while looping ...
^C
$
```

Let's understand exactly what happened here. First, the `while` loop executed the `true` command and evaluated its exit status. Since the exit status of `true` is always 0, it executed the body of the loop (`echo 'Hi, while looping ...'`). Then it went back for another cycle of the same. Because the `true` command always runs with success, this created an infinite loop (which was broken with Ctrl-C). Since `true` is a command, you can replace it with any command. For example:

```
$ while /bin/echo 'ECHO'; do echo 'Hi, while looping ...'; done
ECHO
Hi, while looping ...
ECHO
Hi, while looping ...
ECHO
Hi, while looping ...
^C
$
```

Thus, this `while` loop merely alternates the execution of the two `echo` commands: `/bin/echo`, the external executable, and `echo`, the Bash built-in.

As you might suspect, the `while` construct can accept a command list, and in such a case, it would proceed to execute the body of the loop based on the exit status of the last command in the list. In other words, the general form of the `while` loop is as follows: `while CMD1; CMD2; ... CMDN; do CMDN+1; CMDN+2; CMDN+M; done`. For example:

```
$ while true; false; do echo 'Hi, looping ...'; done
$
```

In this example, the body of the loop is not executed because the last command is `false` (which always fails). The `until` loop works similarly:

```
$ until false; do echo 'Hi, until looping ...'; done
Hi, until looping ...
Hi, until looping ...
Hi, until looping ...
^C
$
```

In the case of the `until` loop, the body of the loop executes as long as the command listed after the keyword `until` is returning a non-zero exit status. Since the command `false` returns a non-zero exit status every time, the above example resulted in an infinite loop. And of course, in the general form, the `until` loop can accept command lists: `until CMD1; CMD2; ... CMDN; do CMDN+1; CMDN+2; CMDN+M; done`.

You may ask, if these loops merely execute two commands (or two command lists), how is it useful in practice at all? The commands that are tested in the loop may depend on some dynamic condition (for example, the number of bytes written to a file, or the type of network traffic and so on). The change in conditions may cause the command to fail or succeed. Also you can modify the Bash variable in the body of the loop, which leads to the use of loops similarly as shown here:

```
$ i=1
$ while [ $i -le 3 ]; do echo $i; i=$((i+1)); done
1
2
3
$
```

Here `((i + 1))` forces Bash arithmetic evaluation, and `$((i + 1))` returns the resulting value; the construct `[ $i -le 3 ]` is a synonym for `test $i -le 3` that performs arithmetic comparison. Note that from the perspective of Bash, this is a command that executes successfully or not:

```
$ i=1
$ [ $i -le 3 ]
$ echo $?
0
$ i=9
$ [ $i -le 3 ]
$ echo $?
1
$
```

This is why the `[ $i -le 3 ]` construct can be used after the `while` keyword, which expects a command (or a command list).

Conclusion

Bash is an independently implemented derivative of the Bourne shell produced by the GNU Project, with enhancements inspired by the C shell and the Korn shell. The original UNIX shell (the Thompson shell) was a simple command interpreter. Subsequently, the Mashey shell and the Bourne shell blended in programming capabilities. Since Bash is a direct descendant of the Bourne shell, it inherited all the key ideas of how a programming environment works. This includes how it blends the programming language with the command interpreter. And for that purpose, UNIX shells have evolved some ingenious solutions.

In Bash, the programming constructs look similar to those found in traditional programming languages. However, how those programming constructs inherently work is quite different. This can be rather confusing to people coming with some knowledge of the traditional programming languages (which is usually the case for Bash users). Here are the three main sources of confusion with Bash programming:

1. The surprising aspect of Bash programming is that the constructs `if`, `while` and `until` evaluate the exit status of a command. Basically these constructs evaluate the following: "is the exit status zero?" By the UNIX convention, the exit status of 0 denotes success, and anything else denotes a failure.

2. The exit status is an integer returned by the executable—think of this as the value returned by the C function `main()`. Note that a program that runs just fine may return a non-zero exit status (I showed an example of this above). However, writing such programs is not recommended. It would break the convention, and it most likely will break other things since the entire environment relies heavily on this convention.

3. A single command can be replaced by a list of commands separated by a semicolon. In such a case, the exit status of a command list is the status returned by the last executed command.

Acknowledgements

My sincere thanks to Chet Ramey for his feedback on the draft of this article. I would also like to thank Isidora C. Likic for checking the text and examples.

Vladimir Likic holds a PhD in bioinformatics, and he has been using UNIX since 1991 and Linux since 1995. Originally from Europe, he lived in the US for a number of years and now calls Australia home. Follow Vladimir on Twitter: @unix_byte.

Resources

Too many articles and books on this topic exist to list in this space, but if you're interested in learning more, we recommend these *Linux Journal* articles (and there are actually too many *LJ* articles to list here as well, but here are some to get you started):

- "Creating the Concentration Game PAIRS with Bash" by Dave Taylor
- "Create Dynamic Wallpaper with a Bash Script" by Patrick Wheelan
- "Developing Console Applications with Bash" by Andy Carlson
- "Hacking a Safe with Bash" by Adam Kosmin
- "Ubuntu Linux and Bash as a Windows Program!" by Dave Taylor
- "Bash Parameter Expansion" by Mitch Frazier
- "Bash Regular Expressions" by Mitch Frazier
- "Bash Extended Globbing" by Mitch Frazier
- "My Favorite bash Tips and Tricks" by Prentice Bisbal
- Parsing an RSS News Feed with a Bash Script" by Jim Hall

Bash Videos:

- "Getting Loopy with Bash: Using for Loops" by Shawn Powers
- "Bash Startup Scripts: bashrc and bash_profile" by Shawn Powers

Getting Started with Rust: Working with Files and Doing File I/O

How to develop command-line utilities in Rust.

By Mihalis Tsoukalos

This article demonstrates how to perform basic file and file I/O operations in Rust, and also introduces Rust's ownership concept and the Cargo tool. If you are seeing Rust code for the first time, this article should provide a pretty good idea of how Rust deals with files and file I/O, and if you've used Rust before, you still will appreciate the code examples in this article.

Ownership

It would be unfair to start talking about Rust without first discussing ownership. Ownership is the Rust way of the developer having control over the lifetime of a variable and the language in order to be safe. Ownership means that the passing of a variable also passes the ownership of the value to the new variable.

Another Rust feature related to ownership is borrowing. Borrowing is about taking control over a variable for a while and then returning that ownership of the variable back. Although borrowing allows you to have multiple references to a variable, only one reference can be mutable at any given time.

Instead of continuing to talk theoretically about ownership and borrowing, let's look at a code example called `ownership.rs`:

```
fn main() {
    // Part 1
    let integer = 321;
    let mut _my_integer = integer;
    println!("integer is {}", integer);
    println!("_my_integer is {}", _my_integer);
    _my_integer = 124;
    println!("_my_integer is {}", _my_integer);

    // Part 2
    let a_vector = vec![1, 2, 3, 4, 5];
    let ref _a_correct_vector = a_vector;
    println!("_a_correct_vector is {:?}", _a_correct_vector);

    // Part 3
    let mut a_var = 3.14;
    {
        let b_var = &mut a_var;
        *b_var = 3.14159;
    }
    println!("a_var is now {}", a_var);
}
```

So, what's happening here? In the first part, you define an integer variable (`integer`) and create a mutable variable based on `integer`. Rust performs a full copy for primitive data types because they are cheaper, so in this case, the `integer` and `_my_integer` variables are independent from each other.

However, for other types, such as a vector, you aren't allowed to change a variable after you have assigned it to another variable. Additionally, you should use a reference for the `_a_correct_vector` variable of Part 2 in the above example, because Rust won't make a copy of `a_vector`.

The last part of the program is an example of borrowing. If you remove the curly braces, the code won't compile because you'll have two mutable variables (`a_var` and `b_var`) that point to the same memory location. The curly braces make `b_var` a local variable that references `a_var`, changes its value and returns the ownership back to `a_var` as soon as the end of the block is reached. As both `a_var` and `b_var` share the same memory address, any changes to `b_var` will affect `a_var` as well.

Executing `ownership.rs` creates the following output:

```
$ ./ownership
integer is 321
_my_integer is 321
_my_integer is 124
my_vector is [1, 2, 3, 4, 5]
a_var is now 3.14159
```

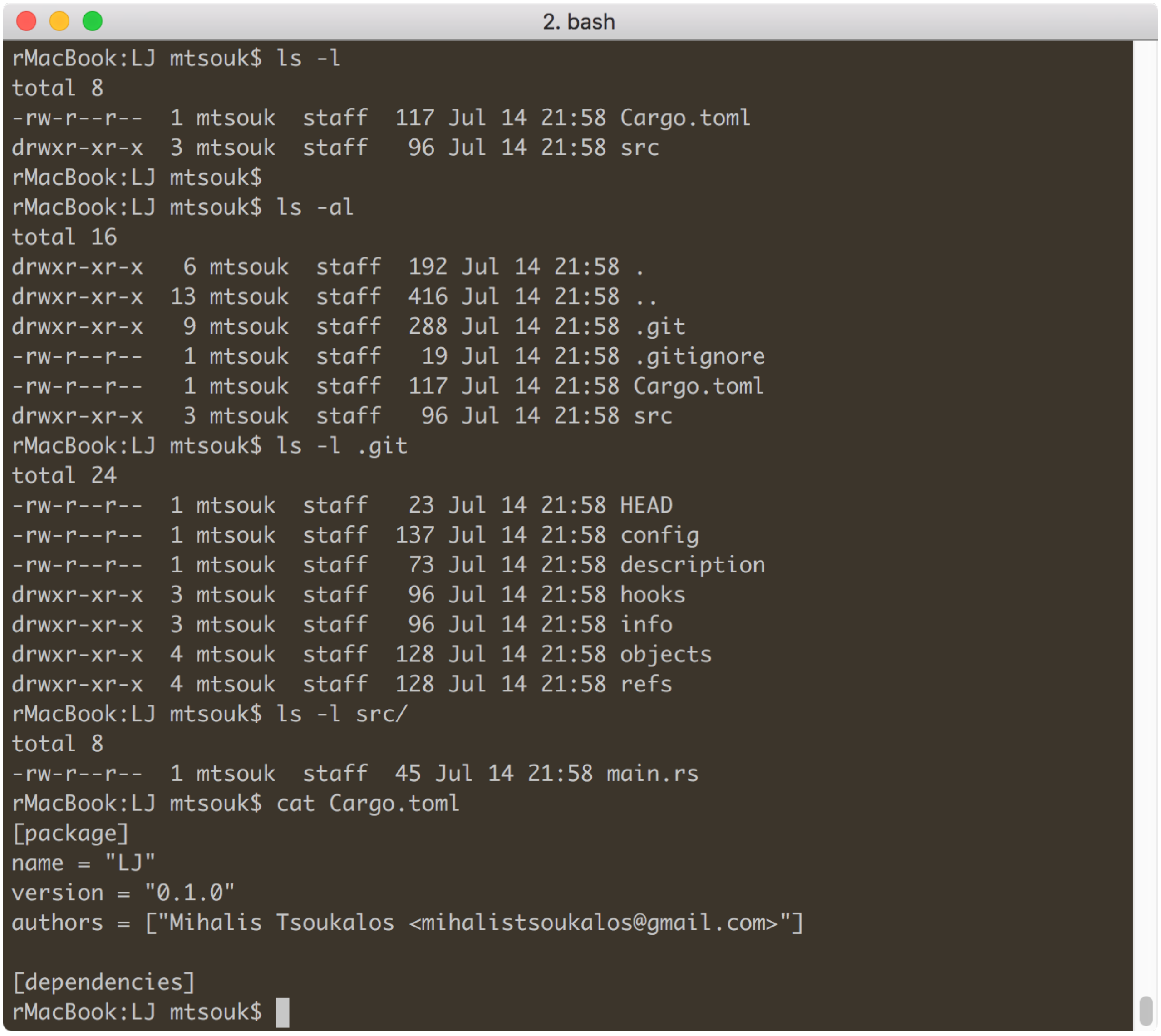
Notice that Rust catches mistakes related to ownership at compile time—it uses ownership to provide code safety.

The remaining Rust code shown in this article is pretty simple; you won't need to know about ownership to understand it, but it's good to have an idea of how Rust works and thinks.

The Cargo Tool

Cargo is the Rust package and compilation manager, and it's a useful tool for creating projects in Rust. In this section, I cover the basics of Cargo using a small example Rust project. The command for creating a Rust project named LJ with Cargo is `cargo new LJ --bin`.

The `--bin` command-line parameter tells Cargo that the outcome of the project will



```
rMacBook:LJ mtsouk$ ls -l
total 8
-rw-r--r--  1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 src
rMacBook:LJ mtsouk$
rMacBook:LJ mtsouk$ ls -al
total 16
drwxr-xr-x   6 mtsouk  staff  192 Jul 14 21:58 .
drwxr-xr-x  13 mtsouk  staff  416 Jul 14 21:58 ..
drwxr-xr-x   9 mtsouk  staff  288 Jul 14 21:58 .git
-rw-r--r--   1 mtsouk  staff   19 Jul 14 21:58 .gitignore
-rw-r--r--   1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x   3 mtsouk  staff   96 Jul 14 21:58 src
rMacBook:LJ mtsouk$ ls -l .git
total 24
-rw-r--r--  1 mtsouk  staff   23 Jul 14 21:58 HEAD
-rw-r--r--  1 mtsouk  staff  137 Jul 14 21:58 config
-rw-r--r--  1 mtsouk  staff   73 Jul 14 21:58 description
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 hooks
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 info
drwxr-xr-x  4 mtsouk  staff  128 Jul 14 21:58 objects
drwxr-xr-x  4 mtsouk  staff  128 Jul 14 21:58 refs
rMacBook:LJ mtsouk$ ls -l src/
total 8
-rw-r--r--  1 mtsouk  staff  45 Jul 14 21:58 main.rs
rMacBook:LJ mtsouk$ cat Cargo.toml
[package]
name = "LJ"
version = "0.1.0"
authors = ["Mihalis Tsoukalos <mihalistsoukalos@gmail.com>"]

[dependencies]
rMacBook:LJ mtsouk$
```

Figure 1. Using Cargo to Create Rust Projects

be an executable file, not a library. After that, you'll have a directory named LJ with the following contents:

```
$ cd LJ
$ ls -l
total 8
-rw-r--r--  1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 src
$ ls -l src/
total 8
-rw-r--r--  1 mtsouk  staff  45 Jul 14 21:58 main.rs
```

Next, you'll typically want to edit one or both of the following files:

```
$ vi Cargo.toml
$ vi ./src/main.rs
```

Figure 1 shows all the files and directories of that minimal Cargo project as well as the contents of Cargo.toml.

Note that the Cargo.toml configuration file is where you declare the dependencies of your project as well as other metadata that Cargo needs in order to compile your project. To build your Rust project, issue the following command:

```
$ cargo build
```

You can find the debug version of the executable file in the following path:

```
$ ls -l target/debug/LJ
-rwxr-xr-x  2 mtsouk  staff  491316 Jul 14 22:02
 ↪target/debug/LJ
```

Clean up a Cargo project by executing `cargo clean`.

Readers and Writers

Rust uses readers and writers for reading and writing to files, respectively. A Rust reader is a value that you can read from; whereas a Rust writer is a value that you can write data to. There are various traits for readers and writers, but the standard ones are `std::io::Read` and `std::io::Write`, respectively. Similarly, the most common and generic ways for creating readers and writers are with the help of `std::fs::File::open()` and `std::fs::File::create()`, respectively. Note: `std::fs::File::open()` opens a file in read-only mode.

The following code, which is saved as `readWrite.rs`, showcases the use of Rust readers and writers:

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("/tmp/LJ.txt")?;
    let buffer = "Hello Linux Journal!\n";
    file.write_all(buffer.as_bytes())?;
    println!("Finish writing...");

    let mut input = File::open("/tmp/LJ.txt")?;
    let mut input_buffer = String::new();
    input.read_to_string(&mut input_buffer)?;
    print!("Read: {}", input_buffer);
    Ok(())
}
```

So, `readWrite.rs` first uses a writer to write a string to a file and then a reader to read the data from that file. Therefore, executing `readWrite.rs` creates the following output:

```
$ rustc readWrite.rs
$ ./readWrite
```

```
Finish writing...
Read: Hello Linux Journal!
$ cat /tmp/LJ.txt
Hello Linux Journal!
```

File Operations

Now let's look at how to delete and rename files in Rust using the code of `operations.rs`:

```
use std::fs;
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("/tmp/test.txt")?;
    let buffer = "Hello Linux Journal!\n";
    file.write_all(buffer.as_bytes())?;
    println!("Finish writing...");

    fs::rename("/tmp/test.txt", "/tmp/LJ.txt")?;
    fs::remove_file("/tmp/LJ.txt")?;
    println!("Finish deleting...");
    Ok(())
}
```

The Rust way to rename and delete files is straightforward, as each task requires the execution of a single function. Additionally, you can see that /tmp/test.txt is created using the technique found in `readWrite.rs`. Compiling and executing `operations.rs` generates the following kind of output:

```
$ ./operations
Finish writing...
Finish deleting...
```

The code of `operations.rs` is far from complete, as there is no error-handling code in it. Please feel free to improve it!

Working with Command-Line Arguments

This section explains how to access and process the command-line arguments of a Rust program. The Rust code of `cla.rs` is the following:

```
use std::env;

fn main()
{
    let mut counter = 0;
    for argument in env::args()
    {
        counter = counter + 1;
        println!("{}: {}", counter, argument);
    }
}
```

Let's look at what's happening in this example. First, it's using the `env` module of the `std` crate, because this is how to get the command-line arguments of your program, which will be kept in `env::args()`, which is an iterator over the arguments of the process. Then you iterate over those arguments using a `for` loop.

Say you want to add the command-line arguments of a program, the ones that are valid integers, in order to find their total. You can use the next `for` loop, which is included in the final version of `cla.rs`:

```
let mut sum = 0;
for input in env::args()
{
    let _i = match input.parse::<i32>() {
      Ok(_i) => {
          sum = sum + _i
      },
      Err(_e) => {
          println!("{}: Not a valid integer!", input)
      }
    };
}

println!("Sum: {}", sum);
```

Here you iterate over the `env::args()` iterator, but this time with a different purpose, which is finding the command-line arguments that are valid integers and summing them up.

If you are used to programming languages like C, Python or Go, you most likely will find the aforementioned code over-complicated for such a simple task, but that's the way Rust works. Additionally, `cla.rs` contains Rust code related to error-handling.

Note that you should compile `cla.rs` and create an executable file before running it, which means that Rust can't easily be used as a scripting programming language. So in this case, compiling and executing `cla.rs` with some command-line arguments creates this kind of output:

```
$ rustc cla.rs
$ ./cla 12 a -1 10
1: ./cla
2: 12
3: a
4: -1
5: 10
./cla: Not a valid integer!
a: Not a valid integer!
Sum: 21
```

Anyway, that's enough for now about the command-line arguments of a program. The next section describes using the three standard UNIX files.

Standard Input, Output and Error

This section shows how to use `stdin`, `stdout` and `stderr` in Rust. Every UNIX operating system has three files open all the time for its processes. Those three files are /dev/stdin, /dev/stdout and /dev/stderr, which you also can access using file descriptors 0, 1 and 2, respectively. UNIX programs write regular data to standard output and error messages to standard error while reading from standard input.

The following Rust code, which is saved as `std.rs`, reads data from standard input and writes to standard output and standard error:

```
use std::io::Write;
use std::io;

fn main() {
    println!("Please give me your name:");
    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(n) => {
            println!("{} bytes read", n);
            print!("Your name is {}", input);
        }
        Err(error) => println!("error: {}", error),
    }

    let mut stderr = std::io::stderr();
    writeln!(&mut stderr, "This is an error message!").unwrap();
    eprintln!("That is another error message!")
}
```

Rust uses the `eprint` and `eprintln` macros for writing to standard error, which is a pretty handy approach. Alternatively, you can write your text to `std::io::stderr()`. Both techniques are illustrated in `std.rs`.

As you might expect, you can use the `print` and `println` macros for writing to standard output. Finally, you can read from standard input with the help of the `io::stdin().read_line()` function. Compiling and executing `std.rs` creates the following output:

```
$ rustc std.rs
$ ./std
Please give me your name:
Mihalis
8 bytes read
Your name is Mihalis
This is an error message!
That is another error message!
```

If you're using the Bash shell on your Linux machine, you can discard standard output or standard error data by redirecting them to /dev/null:

```
$ ./std 2>/dev/null
Please give me your name:
Mihalis
8 bytes read
Your name is Mihalis
$ ./std 2>/dev/null 1>/dev/null
Mihalis
```

The previous commands depend on the UNIX shell you are using and have nothing to do with Rust. Note that various other techniques exist for working with UNIX `stdin`, `stdout` and `stderr` in Rust.

Working with Plain-Text Files

Now let's look at how to read a plain-text file line by line, which is the most frequent way of processing plain-text files. At the end of the program, the total number of characters as well as the number of lines read will be printed on the screen—consider this as a simplified version of the `wc(1)` command-line utility.

The name of the Rust utility is `lineByLine.rs`, and its code is the following:

```
use std::env;
use std::io::{BufReader,BufRead};
use std::fs::File;

fn main() {
    let mut total_lines = 0;
    let mut total_chars = 0;
    let mut total_uni_chars = 0;

    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} text_file", args[0]);
        return;
    }

    let input_path = ::std::env::args().nth(1).unwrap();
    let file = BufReader::new(File::open(&input_path).unwrap());
    for line in file.lines() {
        total_lines = total_lines + 1;
        let my_line = line.unwrap();
        total_chars = total_chars + my_line.len();
        total_uni_chars = total_uni_chars + my_line.chars().count();
    }
    println!("Lines processed:\t\t{}", total_lines);
    println!("Characters read:\t\t{}", total_chars);
    println!("Unicode Characters read:\t{}", total_uni_chars);
}
```

The `lineByLine.rs` utility uses buffered reading as suggested by the use of `std::io::{BufReader,BufRead}`. The input file is opened using `BufReader::new()` and `File::open()`, and it's read using a `for` loop that keeps going as long as there is something to read from the input file.

Additionally, notice that the output of the `len()` function and the output of the `chars().count()` function might not be the same when dealing with text files that contain Unicode characters, which is the main reason for including both of them in `lineByLine.rs`. For an ASCII file, their output should be the same. Keep in mind that if what you want is to allocate a buffer to store a string, the `len()` function is the correct choice.

Compiling and executing `lineByLine.rs` using a plain-text file as input will generate this kind of output:

```
$ ./lineByLine lineByLine.rs
Lines processed:          28
Characters read:          756
Unicode Characters read:  756
```

Note that if you rename `total_lines` to `totalLines`, you'll most likely get the following warning message from the Rust compiler when trying to compile your code:

```
warning: variable 'totalLines' should have a snake case name
such as 'total_lines'
 --> lineByLine.rs:7:6
  |
7 |     let mut totalLines = 0;
  |         ^^^^^^^^^^^^^^
  |
  = note: #[warn(non_snake_case)] on by default
```

You can turn off that warning message, but following the Rust way of defining variable names should be considered a good practice. (In a future Rust article, I'll cover more about text processing in Rust, so stay tuned.)

File Copy

Next let's look at how to copy a file in Rust. The `copy.rs` utility requires two command-line arguments, which are the filename of the source and the destination, respectively. The Rust code of `copy.rs` is the following:

```
use std::env;
use std::fs;

fn main()
{
    let args: Vec<_> = env::args().collect();
    if args.len() >= 3
    {
        let input = ::std::env::args().nth(1).unwrap();
        println!("input: {}", input);
        let output = ::std::env::args().nth(2).unwrap();
        println!("output: {}", output);
        match fs::copy(input, output)
        {
            Ok(n) => println!("{}", n),
            Err(err) => println!("Error: {}", err),
        };
    } else {
        println!("Not enough command line arguments")
    }
}
```

All the dirty work is done by the `fs::copy()` function, which is versatile, as you do not have to deal with opening a file for reading or writing, but it gives you no control over the process, which is a little bit like cheating. Other ways exist to copy a file, such as using a buffer for reading and writing in small byte chunks. If you execute `copy.rs`, you'll see output like this:

```
$ ./copy copy.rs /tmp/output
input: copy.rs
output: /tmp/output
515
```

You can use the handy `diff(1)` command-line utility for verifying that the copy of the file is identical to the original. (Using `diff(1)` is left as an exercise for the reader.)

UNIX File Permissions

This section describes how to find and print the UNIX file permissions of a file, which will be given as a command-line argument to the program using the `permissions.rs` Rust code:

```
use std::env;
use std::os::unix::fs::PermissionsExt;

fn main() -> std::io::Result<()> {
    let args: Vec<_> = env::args().collect();
    if args.len() < 2 {
        panic!("Usage: {} file", args[0]);
    }
    let f = ::std::env::args().nth(1).unwrap();
    let metadata = try!(std::fs::metadata(f));
    let perm = metadata.permissions();
    println!("{:o}", perm.mode());
    Ok(())
}
```

All the work is done by the `permissions()` function that's applied to the return value of `std::fs::metadata()`. Notice the `{:o}` format code in the `println()` macro, which indicates that the output should be printed in the octal system. Once again, the Rust code looks ugly at first, but you'll definitely get used to it after a while.

Executing `permissions.rs` produces the output like the following—the last three digits of the output is the data you want, where the remaining values have to do with the file type and the sticky bits of a file or directory:

```
$ ./permissions permissions
100755
$ ./permissions permissions.rs
100644
$ ./permissions /tmp/
41777
```

Note that `permissions.rs` works only on UNIX machines.

Conclusion

This article describes performing file input and output operations in Rust, as well as working with command-line arguments, UNIX permissions and using standard input, output and error. Due to space limitations, I couldn't present every technique for dealing with files and file I/O in Rust, but it should be clear that Rust is a great choice for creating system utilities of any kind, including tools that deal with files, directories and permissions, provided you have the time to learn its idiosyncrasies. At the end of the day though, developers should decide for themselves whether they should use Rust or another systems programming language for creating UNIX command-line tools. ■

Mihalis Tsoukalos is a UNIX administrator and developer, a DBA and mathematician who enjoys technical writing. He is the author of *Go Systems Programming* and *Mastering Go*. You can reach him at **http://www.mtsoukalos.eu** and @mactsouk.

Resources

- The Rust Programming Language
- Rust Documentation
- The Cargo Book
- Rust Crates
- *Programming Rust* by Jim Blandy and Jason Orendorff, O'Reilly, 2017
- *The Rust Programming Language* by Steve Klabnik and Carol Nichols, No Starch Press, 2018

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

EUROPE

OCTOBER 22-24, 2018 | EDINBURGH, UK

THE PREMIER OPEN SOURCE EVENT OF THE YEAR.

Discover how to best leverage open source technologies at Open Source Summit across AI, Cloud Native, Infrastructure & Automation, Containers, Linux, DevOps and more.

2,000+ Attendees

300+ sessions, workshops, and tutorials

Featured Keynote Speakers:

Patrick Ball
Director of Research, Human Rights Data Analysis Group

Eric Berlow
Co-Founder, Chief Science Officer, Vibrant Data Inc.

Dirk Hohndel
VP & Chief Open Source Officer, VMware

Johanna Koester
Program Director of Developer Technology and Advocacy, IBM

Brenda Romero
Program Director of the MSc in Game Design & Development, Limerick University

Linus Torvalds
Creator of Linux & Git and Fellow, The Linux Foundation

And More!

Use discount code **LJ15** for 15% off

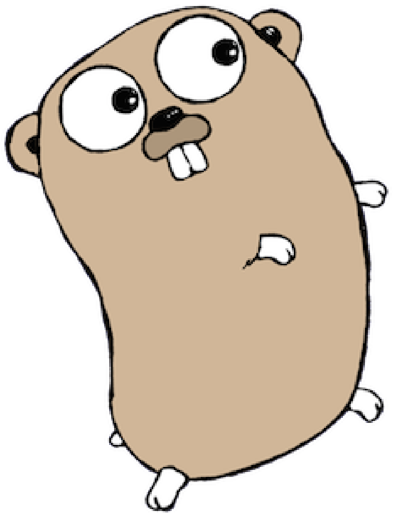
events.linuxfoundation.org/osseu18

Register Now

Introductory Go Programming Tutorial

How to get started with this useful new programming language.

By Jay Ts



You've probably heard of Go. Like any new programming language, it took a while to mature and stabilize to the point where it became useful for production applications. Nowadays, Go is a well established language that is used in web development, writing DevOps tools, network programming and databases. It was used to write Docker, Kubernetes, Terraform and Ethereum. Go is accelerating in popularity, with adoption increasing by 76% in 2017, and there now are Go user groups and Go conferences. Whether you want to add to your professional skills or are just interested in learning a new programming language, you should check it out.

Why Go?

Go has the safety of static typing and garbage collection along with the speed of a compiled language. With other languages, "compiled" and "garbage collection" are associated with waiting around for the compiler to finish and then getting programs that run slowly. But Go has a lightning-fast compiler that makes compile times barely noticeable and a modern, ultra-efficient garbage collector. You get fast compile times along with fast programs. Go has concise syntax and grammar with few keywords, giving Go the simplicity and fun of dynamically typed interpreted languages like Python, Ruby and JavaScript.

Go History

A team of three programmers at Google created Go: Robert Griesemer, Rob Pike and Ken Thompson. The team decided to create Go because they were frustrated with C++ and Java, which through the years have become cumbersome and clumsy to work with. They wanted to bring enjoyment and productivity back to programming.

The three have impressive accomplishments. Griesemer worked on Google's ultra-fast V8 JavaScript engine used in the Chrome web browser, Node.js JavaScript runtime environment and elsewhere. Pike and Thompson were part of the original Bell Labs team that created UNIX, the C language and UNIX utilities, which led to the development of the GNU utilities and Linux. Thompson wrote the very first version of UNIX and created the B programming language, upon which C was based. Later, Thompson and Pike worked on the Plan 9 operating system team, and they also worked together to define the UTF-8 character encoding.

The idea of Go's design is to have the best parts of many languages. At first, Go looks a lot like a hybrid of C and Pascal (both of which are successors to Algol 60), but looking closer, you will find ideas taken from many other languages as well.

Go is designed to be a simple compiled language that is easy to use, while allowing concisely written programs that run efficiently. Go lacks extraneous features, so it's easy to program fluently, without needing to refer to language documentation while programming. Programming in Go is fast, fun and productive.

Let's Go

First, let's make sure you have Go installed. You probably can use your distribution's package management system. To find the Go package, try looking for "golang", which is a synonym for Go. If you can't install it that way, or if you want a more recent version, get a tarball from https://golang.org/dl and follow the directions on that page to install it.

When you have Go installed, try this command:

```
$ go version
go version go1.10 linux/amd64
```

The output shows that I have Go version 1.10 installed on my 64-bit Linux machine.

Hopefully, by now you've become interested and want to see what a complete Go program looks like. Here's a very simple program in Go that prints "hello, world":

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

The line `package main` defines the package that this file is part of. Naming `main` as the name of the package and the function tells Go that this is where the program's execution should start. You need to define a `main` package and `main` function even when there is only one package with one function in the entire program.

At the top level, Go source code is organized into packages. Every source file is part of a package. Importing packages and exporting functions are child's play.

The next line, `import "fmt"` imports the fmt package. It is part of the Go standard library and contains the `Printf()` function. Often you'll need to import more than one package. To import the `fmt`, `os` and `strings` packages, you can type either this:

```
import "fmt"
import "os"
import "strings"
```

or this:

```
import (
    "fmt"
    "os"
    "strings"
    )
```

Using parentheses, `import` is applied to everything listed inside the parentheses, which saves some typing. You'll see parentheses used like this again elsewhere in Go, and Go has other kinds of typing shortcuts too.

Packages can export constants, types, variables and functions. To export something, just capitalize the name of the constant, type, variable or function you want to export. It's that simple.

Notice that there are no semicolons in the "hello, world" program. Semicolons at the ends of lines are optional. Although this is convenient, it leads to something to be careful about when you are first learning Go. This part of Go's syntax is implemented using a method taken from the BCPL language. The compiler uses a simple set of rules to "guess" when there should be a semicolon at the end of the line, and it inserts one automatically. In this case, if the right parenthesis in `main()` were at the end of the line, it would trigger the rule, so it's necessary to place the open curly bracket after `main()` on the same line.

This formatting is a common practice that's allowed in other languages, but in

Go, it's required. If you put the open curly bracket on the next line, you'll get an error message.

Go is unusual in that it either requires or favors a specific style of whitespace formatting. Rather than allowing all sorts of formatting styles, the language comes with a single formatting style as part of its design. The programmer has a lot of freedom to violate it, but only up to a point. This is either a straitjacket or godsend, depending on your preferences! Free-form formatting, allowed by many other languages, can lead to a mini Tower of Babel, making code difficult to read by other programmers. Go avoids that by making a single formatting style the preferred one. Since it's fairly easy to adopt a standard formatting style and get used to using it habitually, that's all you have to do to be writing universally readable code. Fair enough? Go even comes with a tool for reformatting your code to make it fit the standard:

```
$ go fmt hello.go
```

Just two caveats: your code must be free of syntax errors for it to work, so it won't fix the kind of problem I just described. Also, it overwrites the original file, so if you want to keep the original, make a backup before running `go fmt`.

The `main()` function has just one line of code to print the message. In this example, the `Printf()` function from the `fmt` package was used to make it similar to writing a "hello, world" program in C. If you prefer, you can also use this:

```
fmt.Println("hello, world")
```

to save typing the `\n` newline character at the end of the string.

Now let's compile and run the program. First, copy the "hello, world" source code to a file named `hello.go`. Then compile it using this command:

```
$ go build hello.go
```

And to run it, use the resulting executable, named `hello`, as a command:

```
$ hello
hello, world
```

As a shortcut, you can do both steps in just one command:

```
$ go run hello.go
hello, world
```

That will compile and run the program without creating an executable file. It's great for when you are actively developing a project and are just checking for errors before doing more edits.

Next, let's look at a few of Go's main features.

Concurrency

Go's built-in support for concurrency, in the form of *goroutines*, is one of the language's best features. A goroutine is like a process or thread, but it's much more lightweight. It's normal for a Go program to have thousands of active goroutines. Starting up a goroutine is as simple as:

```
go f()
```

The function `f()` then will run concurrently with the main program and other goroutines. Go has a means of allowing the concurrent pieces of the program to synchronize and communicate using *channels*. A channel is somewhat like a UNIX pipe; it can be written to at one end and read from at the other. A common use of channels is for goroutines to indicate when they have finished.

The goroutines and their resources are managed automatically by the Go runtime system. With Go's concurrency support, it's easy to get all of the cores and threads of a multicore CPU working efficiently.

Types, Methods and Interfaces

You might wonder why types and methods are together in the same heading. It's because Go has a simplified object-oriented programming model that works along with its expressive, lightweight type system. It completely avoids classes and type hierarchies, so it's possible to do complicated things with datatypes without creating a mess. In Go, methods are attached to user-defined types, not to classes, objects or other data structures. Here's a simple example:

```
// make a new type MyInt that is an integer

type MyInt int

// attach a method to MyInt to square a number

func (n MyInt) sqr() MyInt {
    return n*n
}

// make a new MyInt-type variable
// called "number" and set it to 5

var number MyInt = 5

// and now the sqr() method can be used

var square = number.sqr()

// the value of square is now 25
```

Along with this, Go has a facility called *interfaces* that allows mixing of types. Operations can be performed on mixed types as long as each has the method or methods attached to it, specified in the definition of the interface, that are needed for the operations.

Suppose you've created types called `cat`, `dog` and `bird`, and each has a method called `age()` that returns the age of the animal. If you want to add the ages of all animals in one operation, you can define an interface like this:

```
type animal interface {
    age() int
}
```

The `animal` interface then can be used like a type, allowing the `cat`, `dog` and `bird` types all to be handled collectively when calculating ages.

Unicode Support

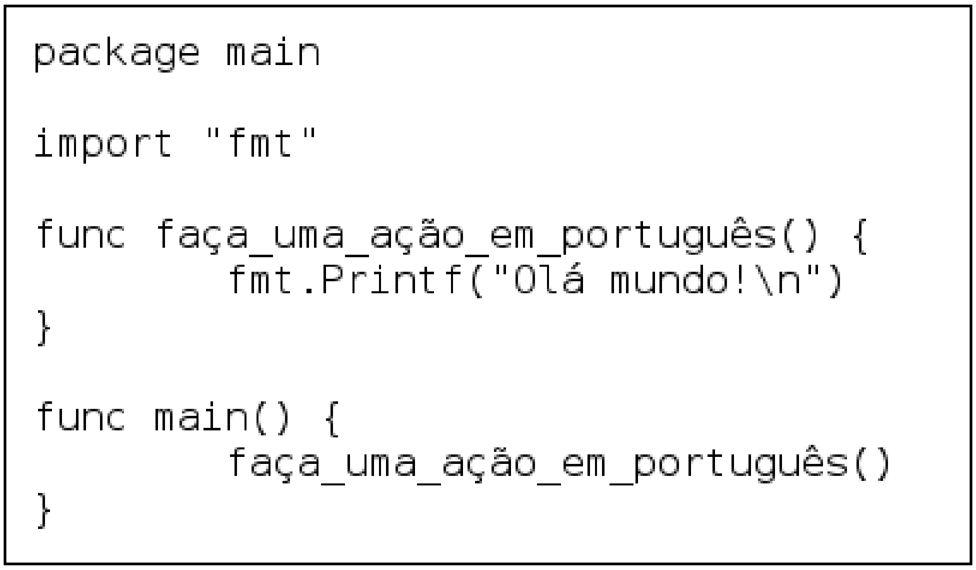
Considering that Ken Thompson and Rob Pike defined the Unicode UTF-8 encoding that is now dominant worldwide, it may not surprise you that Go has good support for UTF-8. If you've never used Unicode and don't want to bother with it, don't worry; UTF-8 is a superset of ASCII. That means you can continue programming in ASCII and ignore Go's Unicode support, and everything will work nicely.

In reality, all source code is treated as UTF-8 by the Go compiler and tools. If your system is properly configured to allow you to enter and display UTF-8 characters, you can use them in Go source filenames, command-line arguments and in Go source code for literal strings and names of variables, functions, types and constants.

In Figure 1, you can see a "hello, world" program in Portuguese, as it might be written by a Brazilian programmer.

In addition to supporting Unicode in these ways, Go has three packages in its standard library for handling more complicated issues involving Unicode.

By now, maybe you understand why Go programmers are enthusiastic about the language. It's not just that Go has so many good features, but that they are all



```
package main

import "fmt"

func faça_uma_ação_em_português() {
        fmt.Printf("Olá mundo!\n")
}

func main() {
        faça_uma_ação_em_português()
}
```

Figure 1. Go "Hello, World" Program in Portuguese

included in one language that was designed to avoid over-complication. It's a really good example of the whole being greater than the sum of its parts. ■

Jay Ts is a software developer, Linux system administrator and electronic designer. He got started with UNIX and the C Programming Language in 1981, and switched from UNIX to Linux in 1996. He is familiar with many operating systems, Linux distributions and programming languages. Jay formerly worked for Caltech, NASA/JPL and the Information Sciences Institute. He currently lives in Sedona, Arizona. Send comments to jay@jayts.com.

Resources

- Official Go Website
- Go Tour
- Go Playground
- Go GitHub Repository
- *The Go Programming Language* by Alan A. A. Donovan and Brian W. Kernighan, Addison-Wesley, 2015

Creating Linux Command-Line Tools in Clojure

Learn how the leiningen utility can help you manage your Clojure projects.

By Mihalis Tsoukalos

This article is a gentle introduction to the Clojure Functional Programming language that is based on LISP, uses the Java JVM and has a handy REPL. And, as Clojure is based on LISP, be prepared to see lots of parentheses!

Installing Clojure

You can install Clojure on a Debian Linux machine by executing the following command as root or using sudo:

```
# apt-get install clojure
```

Finding the version of Clojure you are using is as simple as executing one of the following commands inside the Clojure REPL, which you can enter by running `clojure`:

```
# clojure
Clojure 1.8.0
user=> *clojure-version*
{:major 1, :minor 8, :incremental 0, :qualifier nil}
user=> (clojure-version)
"1.8.0"
user=> (println *clojure-version*)
{:major 1, :minor 8, :incremental 0, :qualifier nil}
nil
```

The first command gets you into the Clojure REPL, which displays the `user=>` prompt and waits for user input. The remaining three commands that should be executed within the Clojure REPL will generate the same output, which, in this example, shows that Clojure version 1.8.0 is being used. So, if you're following along, congratulations! You have just run your first Clojure code!

The leiningen Utility

The first thing you should do after getting Clojure is to install a very handy utility named leiningen, which is the easiest way to use and manage Clojure projects on your Linux machine. Follow the instructions at leiningen.org or use your favourite package manager to install leiningen on your Linux machine. Additionally, if you are using Clojure all the time and working with large Clojure projects, tools like Jenkins and Semaphore will automate your build and test phases and save you lots of time.

After installing leiningen, use the `lein` command (which is the name of the executable file for the leiningen package) to create a new project named hw:

```
$ lein new hw
Generating a project called hw based on the 'default' template.
The default template is intended for library projects,
not applications. To see other templates (app, plugin, etc),
try 'lein help new'.
```

The preceding command will create a new directory named hw that will contain files and other directories. You'll need to make some changes to some of the project files in order to execute the project. First, you'll need to edit the project.clj that can be found inside the hw directory and make it as follows:

```
$ cat project.clj
(defproject hw "0.1.0-SNAPSHOT"
  :main hw.core
  :dependencies [[org.clojure/clojure "1.8.0"]])
```

Then, edit the ./src/hw/core.clj file so it looks like this:

```
$ cat src/hw/core.clj
(ns hw.core)

(defn -main [& args]
  (println "Hello World!"))
```

The ./src/hw/core.clj file is where you can find the Clojure code. Executing the preceding project is as simple as running the `lein run` command inside the directory of the project:

```
$ lein run
Hello World!
```

The first time you execute `lein run`, `lein` might automatically download some files that are required for building the project. Additionally, keep in mind that `lein` can do many more things than what I describe here. The next most important `lein` commands are `lein clean`, which cleans up a Clojure project, and `lein repl`, which starts the Clojure console.

Clojure Data Types

Clojure's philosophy is based on Lisp, which means that Clojure code contains lots of parentheses. Additionally, Clojure is a functional and dynamically typed programming language that has support for concurrent programming, which means Clojure's functions (try to) have no side effects. As the implementation of Clojure is based on the Java Virtual Machine, Clojure data types are Java data types, which means that all Clojure values are in reality references to Java classes. Moreover, most Clojure data types are immutable, which means they can't be changed after they've been created. Finally, Clojure has an unusual way of checking for equality. In order to find out whether two lists are the same, Clojure checks the actual values of the two lists. Most programming languages don't do that because it might be a slow process, especially when dealing with large lists. Nevertheless, Clojure avoids that risk by keeping a hash for each one of its objects and by comparing the hashes of two objects instead of actually visiting all their values. This works as long as the objects are immutable, because if objects are mutable and one of the objects changes, its hash table won't be updated to reflect that change.

In summary, Clojure supports numbers, booleans, characters, strings, nil values, function variables, namespaces, symbols, collections, keywords and vars. A var is one of the mutable Clojure types. A collection can be a list, a hashmap, a vector or a sequence, but lists and hashmaps are the most popular Clojure data types.

But, that's enough of a Clojure introduction; let's start writing real Clojure code.

Working with Data

First, let's look at how to define and populate variables in Clojure as well as how to visit all the elements of a list using the Clojure shell. The Clojure syntax requires that you put the operators as well as the functions in a prefix way, which means that operators are placed before their arguments and not between the arguments (infix). Putting it simply, to calculate the sum of 9 and 4, you would write + 9 4 and not 9 + 4.

Interaction with the Clojure shell starts like this:

```
user=> (- 10)
-10
user=> (- 10 10)
0
user=> (- 10 (+ 5 5) )
0
user=> (/ 10 (+ 5 5) )
1
user=> (println "Hello\nLinux Journal!")
Hello
Linux Journal!
nil
user=> (str "w12")
"w12"
```

This Clojure code does some basic things with numbers first and then with strings. In the first statement, you can see that everything in Clojure must be put in parentheses. The third numeric operation is equivalent to 10 - (5 + 5), which equals zero; whereas the fourth numeric operation is equivalent to 10 / (5 + 5), which equals 1. As you already saw in the Hello World program, the `println` function is used for printing data on the screen; whereas the `str` function can help you convert anything, including numbers, into a string. A good bonus of `str` is that you can use it for concatenating strings when it's called with multiple arguments.

The next interaction verifies that characters in Clojure, which are written as `\a`, `\b` and so on, are not equivalent to Clojure strings, which use double quotes, with a length of 1. However, when you process a single character with `str`, you get a string:

```
user=> (= \a "a")
false
user=> (= (str \a) "a")
true
```

And now, get ready for something more advanced:

```
user=> (map (fn [x] (.toUpperCase x)) (.split
 ↪"Hello Linux Journal!" " "))
("HELLO" "LINUX" "JOURNAL!")
```

The preceding Clojure code does many things. It splits its input string into words and converts each word to uppercase—the good thing is that the way this statement is written in Clojure is natural and easy to read—as long as you start reading it from the right to the left.

The following interaction with the Clojure shell shows how you can work with Clojure maps, which (as you might expect) associate keys with values:

```
user=> (def myMap {:name "Mihalis"
:surname "Tsoukalos"
:livesAt {:country "Greece"
:city "Athens" } } )
#'user/myMap
```

First, you create a new map and assign it to a variable named `myMap`. Notice that `myMap` contains a nested value—that is, a map within a map.

In the next interaction, you'll see various ways to get data from the previous map:

```
user=> (get myMap :country)
nil
user=> (get myMap :name)
"Mihalis"
user=> (myMap :name)
"Mihalis"
user=> (:name myMap)
"Mihalis"
```

```
user=> (get myMap :surname)
"Tsoukalos"
user=> (get-in myMap [:livesAt :country])
"Greece"
user=> (get-in myMap [:livesAt :city])
"Athens"
user=> (get-in myMap [:livesAt :wrong])
nil
```

So, you can get the value of a key using the `get` keyword, and you can travel inside nested values with the `get-in` keyword. Moreover, there are two additional ways to get the value of a key without needing to use the `get` keyword, which are illustrated in the second and the third commands.

Additionally, if a key does not exist, you'll get a `nil` value. Finally, here's how to iterate over all the elements of a list:

```
user=> (def myList (list 0 1 2 3 4 5))
#'user/myList
user=> (doseq [[value index] (map vector myList (range))]
(println index ": " value))
0 :  0
1 :  1
2 :  2
3 :  3
4 :  4
5 :  5
nil
```

So, first you store a list with numbers to the `myList` variable, and then you use `doseq` to iterate over the elements of the list.

Calculating Fibonacci Numbers

This section shows how to define a function in Clojure that calculates natural numbers that belong to the Fibonacci sequence. Create the Clojure project for calculating numbers of the Fibonacci sequence like this:

```
$ lein new fibo
$ cd fibo
$ vi src/fibo/core.clj
$ vi project.clj
```

The contents of src/fibo/core.clj should be this:

```
$ cat src/fibo/core.clj
(ns fibo.core)

(def fib
  (->> [0 1]
    (iterate (fn [[a b]] [b (+ a b)]))
    (map first)))

(defn -main [& args]
  (println "Printing Fibonacci numbers!"))
  (println (nth fib 10))
  (println (take 15 fib))
```

In the aforementioned code, the definition of the `fib` function is responsible for calculating the numbers of the Fibonacci sequence. After that, the main function uses `fib` two times. The first time is to get a specific Fibonacci number, and the second time is to get a list with the first 15 Fibonacci numbers.

Executing the fibo project generates output like the following:

```
$ lein run
55
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
Printing Fibonacci numbers!
```

When you start feeling comfortable with Clojure, try implementing the `fib` function differently because there are many more ways to calculate Fibonacci numbers in Clojure.

Working with Command-Line Arguments

Now, let's look at how to use the command-line arguments of a program in Clojure using a lein project. The steps for creating the "cla" project are as follows:

```
$ lein new cla
$ cd cla
```

First, you should edit src/cla/core.clj to include the actual Clojure code that deals with the command-line arguments of the program. After that, you edit project.clj, and you are done. You can find the Clojure code that actually works with the command-line arguments of the program in the main function that is defined inside src/cla/core.clj:

```
(defn -main [& args] ; Get command line arguments
  (if-not (empty? args)
    (doseq [arg args]
       (println arg))

; In case there are no command line arguments
    (throw (Exception. "Need at least one
 ↪command line argument!"))))
```

The previous Clojure code iterates over the items of the `args` variable using `doseq` and prints each one of its items. Additionally, the last line of code illustrates how to handle exceptions in Clojure. You need that line because `doseq` won't run if the `args` list is empty, which will happen when a program is executed without any command-line arguments. Finally, you can see that comments in Clojure are lines that begin with a semicolon or the part of the line after a semicolon character.

Executing the Clojure project generates output like the following:

```
$ lein run one 2 three -5
one
2
three
-5
```

As you can see, the way to give command-line arguments to a Clojure project is the same as in most programming languages. Note that if you execute `lein run` without giving any command-line arguments, the program will panic and produce lots of debugging output, including the following message:

```
$ lein run
Exception in thread "main" java.lang.Exception: Need at least
one command line argument!,
```

Getting User Input

Apart from using the command line-arguments of a program, there is an alternative way for getting user input, which is during the execution of the program. Here's how to get input from the user in the Clojure shell using the `read-line` function:

```
user=> (def userInput (read-line))
Hello there!
#'user/userInput
user=> (println userInput)
Hello there!
nil
```

The first command uses the `read-line` function to read a line from the user and assigns that line to a new variable named `userInput`; whereas the second command prints the value of the `userInput` variable.

Clojure Macros

Macro definitions look like function definitions, as they have a name, a list of arguments and a body with the Clojure code, and they allow the Clojure compiler to be extended using user code. Generally speaking, there are three circumstances when you need to use macros in Clojure: when you want to execute code at compile time, when you want to use inline code and when you need to access un-evaluated arguments. However, as macros are available only at compile time, it's better to use functions instead of macros when possible.

File Copying in Clojure

Next, here's how to copy a file in Clojure, in case you want to evaluate whether Clojure can be used as a systems programming language like C and Go. As you might expect, you'll use the `lein` utility for generating the project:

```
$ lein new copy
$ cd copy/
$ vi project.clj
$ vi src/copy/core.clj
```

The last two commands signify that you need to change the project.clj and src/copy/core.clj files.

You can find this project's logic in the implementation of the `main()` function:

```
(defn -main [& args]
   (let [input (clojure.java.io/file "/tmp/aFile.txt")
        output (clojure.java.io/file "/tmp/aCopy.txt")]

    (try
      (= nil (clojure.java.io/copy input output))
      (catch Exception e (str "exception: "
 ↪(.getMessage e))))) )
```

As it happens with most programming languages, you can use many techniques in order to copy a file. This example uses the simplest method for copying a file with a single function call. Other techniques include reading the input file all at once and writing it to the output file the same way, and reading the input file line by line and writing to the output file line by line. For reasons of simplicity, the input and output filenames are hard-coded in the project files and are assigned to two variables named `input` and `output`, respectively. After that, a call to `clojure.java.io/copy` creates a copy of the input file. Although this method doesn't require many lines of code, it might not be very efficient when you want to copy huge files or when you want to be able to change some of the parameters of the process.

Executing the project generates no output, but the desired copy of the input file will be created:

```
$ ls -l /tmp/aFile.txt /tmp/aCopy.txt
ls: /tmp/aCopy.txt: No such file or directory
-rw-r--r--  1 mtsouk  wheel  14 Jun 28 10:32 /tmp/aFile.txt
$ lein run
$ ls -l /tmp/aFile.txt /tmp/aCopy.txt
-rw-r--r--  1 mtsouk  wheel  14 Jun 28 10:49 /tmp/aCopy.txt
-rw-r--r--  1 mtsouk  wheel  14 Jun 28 10:32 /tmp/aFile.txt
```

If you want to make your code more robust, you might want to use a `(.exists (io/file "aFile.txt"))` statement to check whether your input file does exist before trying to copy it and a `(.isDirectory (io/file "/a/path/to/somewhere"))` statement to make sure that neither your input file nor your output file are directories.

Listing the Directories and the Files of a Directory

Finally, let's look at how to visit the files and directories that reside in a given directory. You can create the lein project as follows:

```
$ lein new list
$ cd list
```

As expected, you'll need to edit two files from your new project: project.clj and src/list/core.clj. You can find the program's logic in the Clojure code of the `listFileDir` function that is defined in src/list/core.clj:

```
(defn listFileDir [d]
  (println "Files in " (.getName d))
  (doseq [f (.listFiles d)]
    (if (.isDirectory f)
      (print "* ")
      (print "- "))
    (println (.getName f))))
```

Running your lein project generates output like this:

```
$ lein run
Files in  .
- project.clj
- LICENSE
* test
- CHANGELOG.md
* target
- .hgignore
* resources
- README.md
- .gitignore
* doc
* src
```

Conclusion

This article introduces Clojure, which is a very interesting functional programming language with many fans. The tricky thing is that you need to get used to Clojure by writing small Clojure programs in order to realize its advantages. ■

Mihalis Tsoukalos is a UNIX administrator and developer, a DBA and mathematician who enjoys technical writing. He is the author of *Go Systems Programming* and *Mastering Go*. You can reach him at http://www.mtsoukalos.eu and @mactsouk.

Resources

- Clojure Website
- Clojure Documentation
- leiningen Website
- Jenkins
- Semaphore

Decentralized
Certificate Authority
and Naming

Free and open source contributors only:

handshake.org/signup

Review: System76 Oryx Pro Laptop

Can "by hackers, for hackers" sell laptops? System76 sold an Oryx Pro to Rob, and he's here to tell you about it.

By Rob Hansen

I should start by saying that although I'm definitely no newbie to Linux, I'm new to the world of dedicated Linux laptops. I started with Linux in 1996, when Red Hat 4.0 had just adopted the 2.0 kernel and Debian 1.3 hadn't yet been released. I've run a variety of distros with varying degrees of satisfaction ever since, always looking for the Holy Grail of a desktop UNIX that just plain worked.

About 15 years ago after becoming frustrated with the state of Linux on laptop hardware (in a phrase, "nonexistent hardware support"), I switched my laptops over to Macs and didn't look back. It was a true-blue UNIX that just plain worked, and I was happy. But I increasingly found myself frustrated by things I expected from Linux that weren't available on macOS, and which things like Homebrew and MacPorts and Fink could only partly address.

My last MacBook Pro is now four years old, so it was time to shop around again. After being underwhelmed by this generation of MacBooks, I decided to take the risk on a Linux laptop again.

Oh my, an awful lot has changed in 15 years!

System76

System76 is a Denver-based firm with a "by hackers, for hackers" ethos. It's not the first outfit to have tried to deliver on this promise, nor will it be the last. It follows in a long line pioneered by Red Hat and VA Research, and it will continue in the future with businesses yet to be founded. At this moment in history though, System76 seems to be doing a pretty good job of maintaining that standard.

Inquiries

My initial contact with System76 came by visiting the website and requesting a quote for one of its third-generation Oryx Pro models. The sales staff were responsive, polite and didn't seem to have their personalities obliterated into uniform perfection like the Stepford Salesforce of Lenovo or Dell. I also never caught a whiff of a hard sell from any of them. On three occasions just before being able to put down my hard-earned dinero on an Oryx Pro, my life went sideways, and my laptop fund went to pay for strange emergencies that arose out of nowhere, but the System76 sales staff were cheerfully uncaring about this. The impression I got was they believed they knew were going to miss a sale right then, but whether they missed it forever depended on how they behaved in that instant. It's an enlightened view from which more vendors could stand to learn.

Sales

At last, my laptop fund regenerated, and there were no emergencies on the horizon. I visited the System76 site again and discovered in the intervening months that a new generation of the Oryx Pro had been announced, and the first 100 pre-orders would receive some nice swag. Judging from the swag I received (mostly a nice-looking promotional poster that's actually worth framing and putting on your cube wall), I was one of the first 100. I placed my order May 7, 2018, and was told shipments would begin the first week in June.

A couple minor problems arose. The first was that, as tends to happen with pre-orders, the new units were not available quite on schedule. The second was that System76 wasn't proactive about informing me of the delays. I was originally cited a delivery date of June 11th, but I didn't receive mine until June 15th. In the grand scheme of things, this was a minor issue—I've had far, far worse delays from much bigger vendors. Still, the fact that it was on me to ask about shipments, instead of them telling me there would be a four-day delay was an unforced error on their part.

Price

It's sweet hardware, and it's priced like sweet hardware. My laptop came in at $2,704 (including expedited shipping).

system76
Unleash your potential.
TO UNLEASH POTENTIAL: open box, lift platform, and flatten.
DO NOT EMPLOY SHARP OBJECTS. SERIOUSLY.

Figure 1. Unboxing, Part 1

Figure 2. Unboxing, Part 2

Figure 3. Unboxing, Part 3

Figure 4. Unleashed!

Hardware

The new fourth-generation Oryx Pro is what was only a year ago called a "desktop replacement". But that class of computers refers to boat anchors that were unpleasant to lug around in a messenger bag, not a slim machine that's approaching the dimensions of a MacBook. At 15″ wide, 10″ high, and 3/4″ deep, weighing less than 4.5 pounds, it's closer to an airweight than a boat anchor. (That's 38cm by 26cm by 2cm and just under two kilos, for you nerds out there in civilized countries that use proper measurements.) And packed into this form factor is a 4.1GHz i7-8750H with six cores and 12 threads, 32 gigs of DDR4 RAM at 2400MHz, a half-terabyte NVMe SSD, and—Maestro, cue the drumroll—an 8GB NVIDIA GTX 1070 with a 15.6" 4K HiDPI display. This thing makes the Mac Retinas look jagged, and that's not something I ever thought I'd say.

The usual other things round it out. For video, it has HDMI and two DisplayPort 1.3 outputs, two USB 3.1 Type-C connectors, two USB 3.0 Type-A connectors (one of them powered), audio and mike jacks, gigabit Ethernet and 802.11ac WiFi up to 867Mbps. Oh yes, and Bluetooth. On the 17″ models, you also can get Thunderbolt 3, but this isn't an option on my 15″ model. All of these are pretty much what you should expect on any modern laptop, really.

The keyboard is what the kids today call a chiclet, but it has absolutely nothing in common with the chiclet keyboards I remember. People hated the IBM PCjr chiclet keyboard, but this one is as comfortable as any laptop keyboard I've ever used. The scissor switches are responsive, and the keyboard itself is nicely backlit by a rainbow of LED lights. It's a very nearly full keyboard too, with a full-size numeric keypad. Some buttons are combined with others and accessed via function key-presses, but that's to be expected. The keyboard gets high marks.

Mouse support is provided via a trackpad with two buttons, which is one of the very few mis-designs in the hardware. Although two-button mice are better-understood by casual and business users, a lot of hard-core Linux hackers like the third mouse button—myself included. The screen bezel is a little larger than I'm accustomed to seeing on a modern laptop, but if that's the price I pay for this crisp 4K HiDPI display, I consider that affordable.

There's an integrated webcam that works well out of the box with Google Hangouts, Google Meet and Skype. I stopped testing it at that point and reached for a piece of electrical tape to cover it, and while cutting off a piece of tape, I realized the final mis-design. In 2018, when we're all so keenly aware of our privacy and how malware can hijack a webcam, all vendors should place sliding apertures over their webcams. Getting video should never just be about turning it on in software. There also should be a physical action performed by the user involved—something as simple as sliding away a cover.

Bezel, no third mouse button, no webcam physical aperture—if those are my only complaints about the hardware, I think System76 is doing a pretty good job.

Software

The Pop!_OS is Ubuntu with a skin job, but it's a pretty nice skin. Scratch the surface, and you can find standard GNOME underneath, which is in my mind a positive thing—all those skills you've developed on other distros will transfer over to Pop!_OS nicely. The app store takes visual inspiration from Apple's, but it could stand some better curation. Clicking on the "Games" category gives you a nice list of them, but there's no facility to read user reviews or choose what kind of games you're interested in. Although graphically it's heads and shoulders beyond the package tools of yesteryear, usability-wise it still could stand improvement. Given System76 has committed to making Pop!_OS a first-class hacker distro, I suspect the app store experience is pretty low on the list of priorities—but really, it would be such an easy way to distinguish it from Ubuntu and its other derivatives.

My biggest complaint with Pop!_OS is that it's almost a dark theme but isn't. "Dark muddy" might be a better way to describe the color scheme.

Still, as mentioned earlier, it's all GNOME under the hood, so you can install whatever theme you're accustomed to.

As far as development tools, it seems to all be standard Ubuntu 18.04 repositories, so I won't rehash it except to say that it offers what you expect: GCC 7.3 and 8.0.1, GNAT, Golang, OpenJDK, Mono and the like. The Mono libraries are out of date (4.8, whereas the current is 5.12), but that's on Ubuntu, not Pop!_OS.

Sound and Video

Sound-wise, the Oryx Pro is a little bit of a letdown. There's been so much good stuff that describing the speakers as mediocre feels like a criticism. They're not bad speakers, mind you, they're just not going to impress you much. It's a laptop. It's really, really hard to put good speakers in a laptop. I compromise with a USB headset and everything's great. I've also had fine results with a pair of external USB speakers.

Video-wise, the Oryx Pro is a docile little lamb up until it turns into the Stay-Puft Marshmallow Man and starts stomping New York flat. It ships with two different video chipsets: one an onboard low-power set by Intel and the other the aforementioned NVIDIA GTX 1070 with 8GB RAM. When you engage this monster, this machine stops being a laptop. I speak from experience. An hour of it in my lap was enough to leave my left thigh with first-degree burns. You're aware it's hot, but you tell yourself that you can ignore it. Then you shut down an hour later, look at your leg and wish you hadn't ignored it. The price of machismo, I guess.

The Intel chipset is sufficient for pretty much anything short of intensive 3D, 4K gaming or mining cryptocurrency. If you want to use the HDMI or DisplayPort external jacks, you'll need to switch to the NVIDIA chipset. Switching between chipsets requires a reboot and a surprisingly long wait. My suspicion is some firmware is getting flashed somewhere. By "surprisingly long wait", I mean that I've seen it take up to 20 seconds more to reboot on a chipset switch than to reboot without a chipset switch.

Battery Life

Power is supplied by a pretty standard brick that ends in the expected barrel plug. The trend nowadays is for laptops to be powered by USB-C or Thunderbolt, but really, I don't care much about that. What I care about is whether the vendor-supplied power cable is long enough to be useful, and there we're on good ground. This contributes to the overall weight, of course, which is why so many vendors are intent on giving you power cables that aren't long enough to let you be more than three feet from a socket. System76 is having none of that: you've got about ten feet of distance to work with.

According to System76, the Oryx Pro's battery stores 55 watt-hours (~200 kilojoules) of energy. That's the good news. The bad news is twofold: one, power draw is significantly higher than I'd expect, and two, the onboard battery monitor is completely useless.

I've been composing this article on my System76 laptop in battery-saver mode. The screen is at minimum brightness, the graphics are being provided by the Intel chipset, and I'm avoiding anything that's especially power-hungry. Still, after just 70 minutes, I've dropped from a 98% charge to a 60% charge—assuming I can trust the battery monitor, which I really can't. 70 minutes ago it told me I had 92 minutes of charge remaining; now it tells me I have two hours and seven minutes.

Whatever. The GNOME battery applet always has been painfully inaccurate, in my experience, and that's on GNOME, not System76.

Here's what I can tell you: running purely on battery on a power-saving profile and reducing my power-hungry apps, I've run this laptop for three and a half hours before going dry. That's a significant step below what Apple's getting with its MacBook line. I hope System76 invests in improving Linux and GNOME's power infrastructure, because we can do better than this.

Like other vendors, System76 has done away with the user-swappable battery pack. It used to be that if I need more battery life I could carry a spare battery, but apparently that ship has sailed.

Are we done with this? Good. Hold on while I find a socket, I'm going to go plug this in.

Support

Here's something that might surprise you: my laptop is defective.

Yes, it's defective. That's not unheard of in first-shipped units. Every week or so, it'll spontaneously reboot due to a hardware fault. These reboots are infrequent enough that it's not severely impacting my work, but it still needs addressing, and that's given me a good opportunity to explore System76's support offerings.

Let me give the bottom line first: they're human beings and they care. That's both good news and bad news. It's good news, because human beings who care are so much better than all other alternatives that it's like comparing a supernova to a firecracker. It's bad news, because for things really to get screwed up, you need the involvement of people who are so fervently committed to getting things right, they don't notice they're digging the hole deeper.

I reported my first bout of reboots, along with a copy of my system log for 30 seconds prior to reboot, via the web page the afternoon of June 20th. A few minutes after noon the next day, System76 had approved a no-questions-asked return. On June 22, a customer service rep named Aaron told me "We are shipping your replacement part and will provide you with a tracking number as soon as it is available."

Remember how earlier on when shipments were delayed they didn't inform me about it? Yeah, that happened again. On June 26th, I asked them, "Where is this laptop? I've received no tracking information for a product you said was shipping four days ago."

About an hour later, Emma informed me, "The replacement laptop will take some time to ship, because we are out of stock and awaiting the 4k display, which is expected to arrive the week of July 10th. We are sorry for the delays. We were just notified about the delays and apologize for this inconvenience."

On June 22nd, I was told it was shipping, not "we will ship it as soon as new stock comes in", but that it was shipping. Then, after it became clear there was a delay in new arrivals, they didn't reach out to let me know. Instead, I found out four days later that I wouldn't be receiving my replacement for two weeks.

I complained loudly. Carl, the head honcho at System76, responded to me directly and politely. He took responsibility for the error. System76 assures me it has changed the response system so the company no longer will be sending "we are shipping" notifications ahead of, well, you know, systems actually shipping.

Let me make it clear, I believe Carl. I also think Emma and Aaron and everyone else I've interacted with are good people who genuinely want to deliver the best user experience possible. I don't think my experience with System76 represents its character as a company, except insofar as it represents a company going through growing pains as it adjusts to a level of demand it wasn't expecting.

And really, for how sweet this hardware is, I completely understand the company getting swamped.

The final question is, "if I had the $2,704 to spend again, would I be better served with an System76 Oryx Pro, a MacBook Pro or a Dell XPS?" And on balance, even taking into account the support growing pains, I can say without a shadow of doubt, I would give my money to System76 again.

And I'll also still be pestering System76 to do better. Because once the support infrastructure is cleaned up, believe you me, System76 is going to be giving everybody else in the Linux laptop space a run for their money.

The Takeaway

Pluses:

- A desktop replacement laptop in a near-MacBook form factor.
- i7-8750H with six cores and 12 threads.
- Up to 32GB RAM, and a wide variety of HD options including large NVMe SSDs.
- 55Wh battery, ~3-hour life under real-world conditions.
- Pop!_OS is a nice-looking Ubuntu 18.04 derivative.
- GTX 1070 and Intel GPUs.
- Backlit near-full-size keyboard with numeric keypad.
- Lots of USB ports, including two USB-3.1 Type-Cs.
- Thunderbolt on the 17" model.
- 15" models offer 4K HiDPI displays, which are amazingly crisp.

Minuses:

- Sales and support departments are experiencing growing pains.

- No third button on trackpad.
- No physical aperture on webcam.
- Screen bezel slightly larger than expected.
- Laptop gets dangerously hot when the GTX 1070 kicks in.

Recommendations:

- If you've got the money, this is the best thing I've found for dedicated Linux laptops.
- Be patient with System76's staff. They're having growing pains.
- Tell them I sent you. ■

Rob Hansen (@robertjhansen on Twitter) started using Linux in October 1996 and hasn't looked back since. He graduated from Cornell College in 1998, went on to graduate school at the University of Iowa, and he continues to promise his family he'll someday finish his PhD. Until then, he's saving the world with IronNet Cybersecurity where he enjoys curious pastimes like speaking about himself in the third person while not speaking for his employer.

3D-Printed Firearms Are Blowing Up

What's the practical risk with 3D-printed firearms today? In this opinion piece, Kyle explores the current state of the art.

By Kyle Rankin

If you follow 3D printing at all, and even if you don't, you've likely seen some of the recent controversy surrounding Defense Distributed and its 3D-printed firearm designs. If you haven't, here's a brief summary: Defense Distributed has created 3D firearm models and initially published them for free on its DEFCAD website a number of years ago. Some of those 3D models were designed to be printed with a traditional home hobbyist 3D printer (at least in theory), and other designs were for Defense Distributed's "Ghost Gunner"—a computer-controlled CNC mill aimed at milling firearm parts out of metal stock. The controversy that ensued was tied up in the general public debate about firearms, but in particular, a few models got the most attention: a model of an AR-15 lower receiver (the part of the rifle that carries the serial number) and "the Liberator", which was a fully 3D-printed handgun designed to fire a single bullet. The end result was that the DEFCAD site was forced to go offline (but as with all website take-downs, it was mirrored a million times first), and Defense Distributed has since been fighting the order in court.

The political issues raised in this debate are complicated, controversial and have very little to do with Linux outside the "information wants to be free" ethos in the community, so I leave those debates for the many other articles on this issue that already have been published. Instead, in this article, I want to use my background as a hobbyist 3D printer and combine it with my background in security to build a basic risk assessment that cuts through a lot of the hype and political arguments on all sides. I want to consider the real, practical risks with the 3D models and the current

Ghost Gunner CNC mill that Defense Distributed provides today. I focus my risk assessment on three main items: the 3D-printed AR-15 lower receiver, the Liberator 3D-printed handgun and the Ghost Gunner CNC mill.

3D-Printed AR-15 Lower Receiver

This 3D model was one of the first items Defense Distributed shared on DEFCAD. In case you aren't familiar with the AR-15, its modular design is one of the reasons for its popularity. Essentially every major part of the rifle has numerous choices available that are designed to integrate with the rest of the rifle, and you can find almost all of the parts you need to assemble this rifle online, order them independently, and then build your own—that is, except for the lower receiver. That part of the rifle is what the federal government considers "the rifle", as it is the part that's stamped with the serial number that uniquely identifies and registers one particular rifle versus all of the others out there in the world. This part has restrictions like you would find with a regular rifle, revolver or other firearm.

The fact that the lower receiver gets the serial number is what makes a 3D-printed lower receiver so controversial, because if you can print your own, you can attach it to all of the other rifle parts you purchased online and assemble a rifle that has no serial number. The concern is that people will buy a 3D printer and create an AR-15 rifle that can't be traced.

If you haven't done much 3D printing, you may not know that just because a 3D model for a part exists, it doesn't mean you can print it. Most of the designs offered by sites like Thingiverse were created by 3D modelers who understand the limitations of home 3D printers (like needing to print support material if you print part of an item over the open air), and they design their parts accordingly. The AR-15 lower receiver never was designed to be printed on a 3D printer, and it turns out that this model is a particularly difficult one to print due to the various overhangs and other complex parts of the model. A number of tech articles have been published in which the authors attempt to describe how to print out the lower receiver, and the end result tends to be that it's technically possible to do so on a home 3D printer if it has high enough tolerances, but that even after you deal with removing all of the support material, you still have to spend quite a bit of time cleaning up the part just to make it fit, much less work well, in a real AR-15.

If you look online, you'll also find some video tests of this lower receiver showing that if you do get things to fit and file down the part properly, eventually you may get a receiver that can handle at least a few hundred rounds before it breaks. Note that this is with a well-calibrated and high-quality 3D printer with high-quality plastic. Someone who isn't well versed in the hobby likely will create a part with poor layer adhesion, poor tolerances and poor-quality plastic that won't be nearly as durable.

So what's the risk with this part? The risk is that after investing hundreds of dollars in a 3D printer, many hours of effort in printing and cleaning up the part, and decent expertise into AR-15 design in order to assemble the rifle from scratch, someone *could* create an AR-15 rifle without a serial number. Alternatively, for less than $50, someone could buy a brand-new, lower receiver made out of metal, and instead of using a cheap metal file to painstakingly shape a plastic lower receiver into the right shape, one could just quickly file away the serial number on the metal part.

In the end, any reasonable criminals who wanted to build an untraceable AR-15 wouldn't go to all of the trouble to 3D-print one. Instead, they would just buy a new or used, off-the-shelf, cheap lower receiver (either legally or illegally) and remove the serial number. It's much cheaper, faster and simpler, and it results in a much stronger part, so the threat from a 3D-printed lower receiver in my mind is pretty minimal.

The Liberator

The next controversial part from Defense Distributed that made the news was "The Liberator", which is the first fully 3D-printed handgun. Unlike with the AR-15 lower receiver, the controversy around this firearm was less the concern that it was untraceable, but more the concern that because it was almost 100% plastic (the firing pin is a nail from a hardware store, and Defense Distributed accounted for adding a sheet of metal inside the plastic to comply with legislation prohibiting firearms that defeat a metal detector), in theory, someone could print one of those firearms and get it past a metal detector.

There's a few things to know about the Liberator. First, the firearm can fire only a single round. Second, there are examples online of people who have printed out a Liberator successfully and tested it. The results were that it did indeed fire, although in some cases, the firearm itself exploded instead. Printing durable parts with a hobbyist 3D printer requires a certain level of 3D-printing expertise, quality plastic and a well-calibrated printer.

When you fire a gun, you essentially create an explosion inside the firearm. The firearm is designed to contain that explosion without breaking, and because the firearm doesn't break apart, all of that explosive force propels the bullet through the barrel and out of the gun. When you create a 3D-printed part, you are creating it out of layers of melted plastic. If you calibrate your temperatures and printer well, each layer should melt into the previous layer and stick together. Where those layers join is still a potential weak spot, however, and it just takes a slightly cooler print head to result in bad layer adhesion and a weak part.

So, is the Liberator a practical risk? The typical concern seems to be about some kind of terrorist who is able to smuggle an untraceable gun past security and onto a plane or inside a secured building. Practically speaking, do you think any reasonable attackers would want to be limited to a single shot, or are they going to carry a duffel bag full of these things? Otherwise, what real damage could an attacker do with just a single round? With air marshals frequently flying undercover on planes, would an attacker risk hijacking a plane with a single shot knowing someone might be onboard with a real firearm? The same risk goes for any secured building. Of course, I'm not even factoring in the fact that such a handgun isn't going to be very accurate, and that there's a reasonable chance it will blow up in the attacker's hand instead of firing. Any criminal who would want to use a firearm for a terrorist act would not choose such a risky, flimsy, single-shot device.

Ghost Gunner CNC Mill

The final item to consider is the Ghost Gunner CNC mill. This is a full-featured computer-controlled mill that can take a block of metal, accept an uploaded design and then mill out a perfect metal part. The current Ghost Gunner 2 mill costs around

$2,000, but it requires purchasers to get their own jig sets and, of course, set up the mill itself. Once that's done, they can send designs for an AR-15 lower receiver as well as a frame for a 1911 pistol and mill them out of metal stock. After a bit of clean up, they have a perfect and strong part suitable for replacing any part they otherwise could purchase from a gun dealer.

Like with the 3D-printed AR-15 lower receiver, the controversy with the Ghost Gunner mill is that you can use it to create a part without a serial number. Of course, gunsmiths have long been allowed to make their own rifles and pistols at home legally without serial numbers for personal use, so that part is nothing new. The main controversy here is the fact that you don't need the same level of machinist skills with a Ghost Gunner, because the computer takes care of all of the precision cuts you'd otherwise have to do yourself with a traditional mill. Also unlike the 3D-printed part, this is a real, strong metal part with high tolerances that doesn't require a lot of fine adjustments after it's made.

So let's discuss the practical risk with the Ghost Gunner mill. The first concern presumably is that anyone could purchase a mill and then create an untraceable rifle or handgun. Like with the 3D-printed part, however, do you think a criminal who wanted one or two untraceable guns would go to the trouble of buying a $2,000 machine, buy all of the jigs, set it up, source metal stock and then print the part, when they could buy a new lower receiver for $50 and remove the serial number? I guess after around the 50th lower receiver they would break even on the Ghost Gunner, jigs and metal stock.

The next concern, then, isn't over someone creating a single untraceable firearm but instead mass-producing them. I imagine someone who wants to create an illegal, under-the-radar firearm business could use a Ghost Gunner to do that, but then again, they also could just hire a gunsmith (or learn how to do it themselves) and do it with a traditional mill. In either case, the process is relatively slow compared to normal mass production. The concern here clearly isn't with large-scale arms control, as any big player (drug cartels and the like) would have the resources to buy firearms en masse and wouldn't want to bother with the slow, one-part-at-a-time process with a Ghost Gunner.

So is there a practical risk with the Ghost Gunner? In my opinion, the risk is relatively low. A criminal who wants an untraceable gun has much better, simpler and cheaper options.

Conclusion

Taking political and other concerns out of the equation and focusing only on the practical risk, my conclusion is that the practical risk posed by 3D-printed firearms is relatively low with the current state of the art. In just about every case, criminals or terrorists who want to use a firearm for their crime have much better alternatives and wouldn't bother with the cost, effort and risk associated with 3D-printed weapons.

Disclaimer

The views and opinions expressed in this article are those of the author and do not necessarily reflect those of *Linux Journal*. ■

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.

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

3-4
NOV

PRESENTING

freenode #LIVE2018

20TH ANNIVERSARY CELEBRATION

BRISTOL, UK

OPENS AT 9AM

The freenode project celebrates its 20th anniversary this year at the second annual freenode #live conference

At We The Curious in Bristol, UK
November 3-4, 2018 — 9am Saturday - 6pm Sunday

Keynote speakers include Bradley M. Kuhn, Chris Lamb, Kyle Rankin, Leslie Hawthorn and VM Brasseur! More to come...

Registration and call for participation is open now at

HTTPS://FREENODE.LIVE

Now Is the Time to Start Planning for the Post-Android World

We need a free software mobile operating system. Is it eelo?

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+.

Remember Windows? It was an operating system that was quite popular in the old days of computing. However, its global market share has been in decline for some time, and last year, the Age of Windows ended, and the Age of Android began.

Android—and thus Linux—is now everywhere. We take it for granted that Android is used on more than two billion devices, which come in just about every form factor—smartphones, tablets, wearables, Internet of Things, in-car systems and so on. Now, in the Open Source world, we just assume that Android always will hold around 90% of the smartphone sector, whatever the brand name on the device, and that we always will live in an Android world.

Except—we won't. Just as Windows took over from DOS, and Android took over from Windows, something will take over from Android. Some might say "yes, but not yet". While Android goes from strength to strength, and Apple is content to make huge profits from its smaller, tightly controlled market, there's no reason for Android to lose its dominance. After all, there are no obvious challengers and no obvious need for something new.

However, what if the key event in the decline and fall of Android has already taken place, but was something quite different from what we were expecting? Perhaps it won't be a frontal attack by another platform, but more of a subtle fracture deep within the Android ecosystem, caused by some external shock. Something like this, perhaps:

> Today, the Commission has decided to fine Google 4.34 billion euros for breaching EU antitrust rules. Google has engaged in illegal practices to cement its dominant market position in internet search. It must put an effective end to this conduct within 90 days or face penalty payments.

What's striking is not so much the monetary aspect, impressive though that is, but the following: "our decision stops Google from controlling which search and browser apps manufacturers can pre-install on Android devices, or which Android operating system they can adopt."

Whether or not you agree with the EU's decision, and assuming that it isn't overturned on appeal, that demand for Google to loosen its control over the Android ecosystem is significant, and it may be the beginning of the end of the Android era as we know it. Even if it isn't, the EU fine is a timely reminder that the moment will, inevitably, come. Google clearly knows that, which is probably why it is developing Fuchsia.

Fuchsia will be open source, made up of a mix of BSD 3 clause, MIT and Apache 2.0 licensed code, but not based on Linux. Significantly, the Fuchsia "book" readme file begins: "Fuchsia is not Linux". If Fuchsia turns into a major project that appears on smartphones and elsewhere, the implications for the Linux community are clearly huge. The Open Source world therefore needs to start thinking about what that will mean for the community—and to start planning for it.

Smartphone manufacturers currently dependent on Android already have back-up plans of varying degrees of seriousness. For example, the Chinese tech giant Huawei, now the number two smartphone manufacturer after Samsung, is developing its own alternative, although there are no details yet. Samsung still has Tizen, for what it's worth. The question is, what kind of insurance policy should the Open Source world be putting in place against the day when Google moves off Android?

Alongside all the previous (failed) attempts to come up with a viable free software smartphone operating system, there's a new option that's well worth a look. It comes from Gaël Duval, who probably is best known as the creator of the Mandrake GNU/Linux distribution in 1998. Based on Red Hat, Mandrake set great store by ease of use, and it was the first of a new generation of distros aimed at ordinary users, which have become commonplace today. His new project is the free software mobile operating system called eelo. Duval says he chose the name in part "because eels are small fish that can hide into the sea. That's perfect for my quest of more privacy". Addressing the woeful lack of privacy that is a by-product of using today's smartphones is a major aim of the project:

> Last year, I decided to leave Apple and Google: I want to free myself from the smartphone duopoly; I want to regain control over my data privacy; I want to protect my freedom.
>
> At first, I thought I would just fork Android, add a better design, remove any Google stuff, select a few privacy-compliant web services and add them to the system.
>
> A little more than 6 months later, I realize that we're building something really, really bigger than I had expected. This is made possible by the tremendous support I'm getting from many people around the world, and by a growing community of eelo contributors.

After a successful crowdfunding campaign, Duval has set up a foundation called

/e/ to support eelo's development. Its website contains details about the mobile operating system roadmap, the applications that are planned and the team behind the project. There's also a "manifesto", which is nothing if not ambitious.

It's still the early days for eelo. However, what sets it apart from previous open-source mobile operating system projects—and what makes it worthy of support by the coding community—is the emphasis on privacy. In this respect, it can be seen as a representative phenomenon of the GDPR-suffused world we now inhabit:

> eelo is committed to providing desirable mobile phones and web-services that respect the user's data privacy. The eelo OS will not send a user's data to eelo, such as his location, his contacts, his agenda, in an exploitable manner. eelo users will be able to use eelo cloud services with the guarantee that their data will be kept private and stored as securely as possible.

That's only an aspiration at the moment, but it's a laudable one. For too long, the Open Source world has been complicit with Google in undermining the privacy and freedom of Android users. It's understandable—Android has helped make Linux the most widely used operating system in the world, overthrowing Windows. Fighting against Google in the early days of smartphones, as it tried to establish Android as an alternative to completely proprietary offerings, would have been quixotic. But the time has come to assert free software's underlying ethical foundation and to move on from an Android world to something better—in all senses. Whether that will be Duval's eelo or something else is a matter for the Open Source community to debate. But it's a debate that we need to have now, as a choice, before it becomes a necessity. ■

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