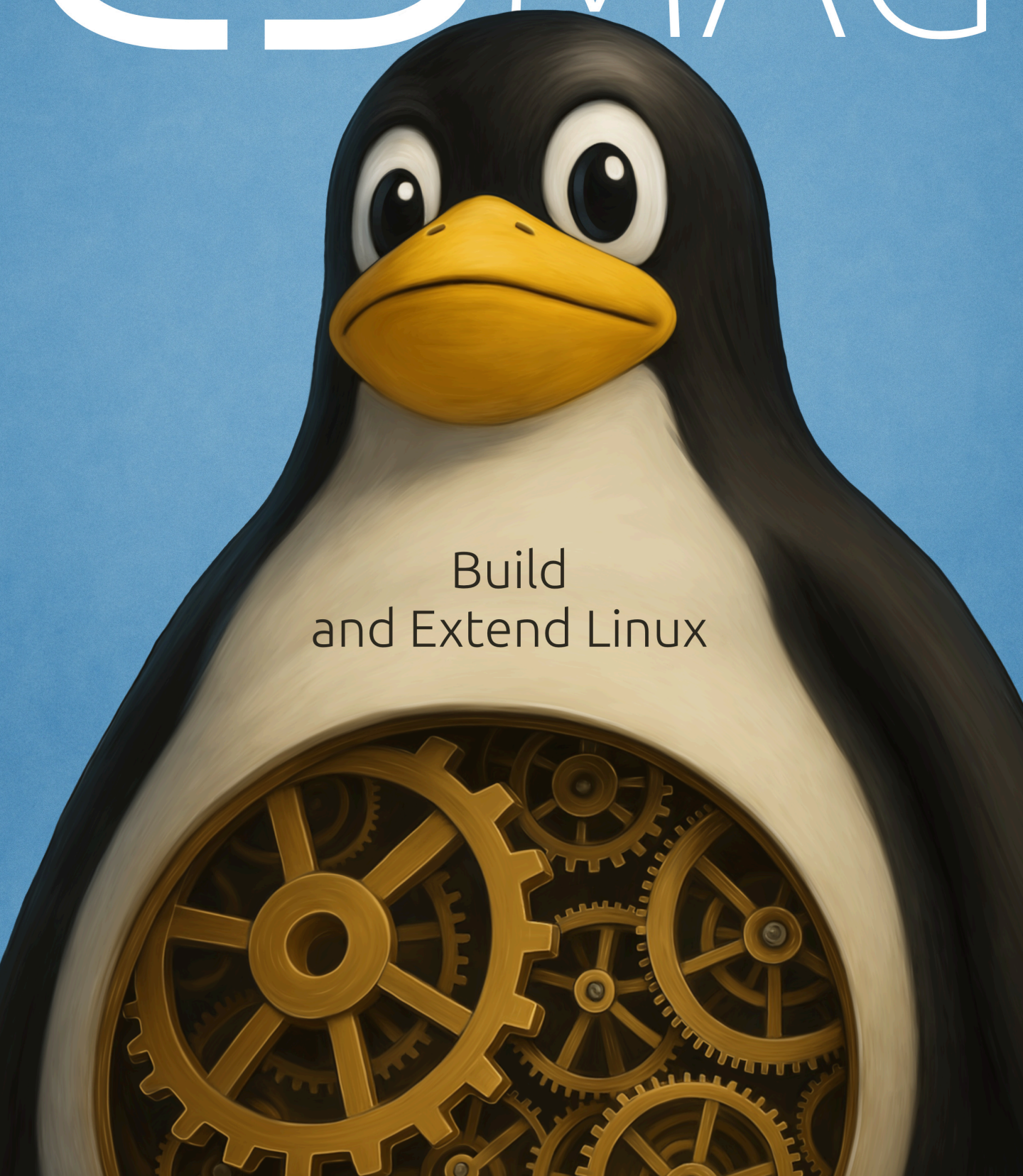
№1

CDMAG



Build
and Extend Linux

CONTENTS

Yet Another Experiment to Run Linux in a Web Browser

The Linux kernel can run anywhere — be it hardware platforms or software ones. In this article, we'll look at yet another attempt to run Linux in a browser.

Ubuntu's Unconventional Solutions

How Canonical's greatest adventure came to an end.

Linux Kernel Hacking Begins

There's a common belief that developing and debugging the Linux kernel is only for the chosen few. Let's debunk this myth.

Only Macros Left Alive

A curious observation — macros are the longest-living creatures in the software world.

Q&A with Evgeny Golyshev

The most interesting questions from **@cusdeb_com**'s followers for November.

Yet Another Experiment to Run Linux in a Web Browser

On **November 1**, **Joel Severin** published the results of porting **Linux** to **WebAssembly** (Wasm) on the Linux kernel mailing lists. The result is a full-fledged port like **ARM64** or **RISC-V** — except that in this case, Linux can run directly in a web browser. These patches aren't expected to be merged anytime soon, since (1) Joel has only ported **Linux 4.6** so far and still needs to adapt the patches for the latest kernel, and (2) it's not yet clear who will maintain this port, as there's still no practical use case for it yet.

WebAssembly

WebAssembly is a *universal* low-level intermediate code for running in-browser applications compiled from various programming languages. Its versatility comes from standardization, which is maintained by the **W3C**. The current version of the standard, **3.0**, was published on **September 17**, **2025**.

WebAssembly can be used for high-performance tasks such as *video encoding*, *audio processing*, *graphics* and *3D rendering*, *game development*, *cryptographic operations*, and *mathematical computations*. It enables code written in compiled languages like **C** and **C++** to run directly in the browser. Thanks to JIT compilation, WebAssembly can achieve performance levels close to native code. C and C++ are typically compiled to WebAssembly using **Emscripten**.

Well-Forgotten Old

The idea of running Linux in a browser's isolated environment may sound new, but it is not. Back in **2011**, **Fabrice Bellard** — a brilliant mathematician and the creator of **QEMU** and **FFmpeg** — implemented an x86 emulator in JavaScript capable of running a minimalist Linux system. He described the project as an experiment to test the limits of modern JavaScript engines and explore optimization techniques for JavaScript code. Later, in **2017**, he announced another project — **VFsync** — along with its companion emulator, **TinyEMU** (formerly known as **RISCVEMU**).

Fabrice Bellard

One of the stated goals of VFsync was to provide a high level of security by running virtual machines. The idea was that each virtual machine environment would be completely isolated from the host system, with the ability to synchronize with external storage — giving users access to their data and workspace from any computer. Before transmission, data was encrypted client-side using AES, and communication was handled over HTTPS. The data could be stored either in the project's cloud infrastructure or on self-hosted components.

In Conclusion

As you can see, Linux can be adapted to run virtually anywhere — and developers never stop finding new ways to make it run in the browser. The current results are impressive, though they remain more of an academic experiment than a practical solution. CD

Ubuntu's Unconventional Solutions

The October Instagram carousel post on the `@cusdeb_com` page announcing Ubuntu 25.10 has surpassed 700k views. Its popularity is largely due to Ubuntu 25.10 being the most Rust-heavy release in the distro's history. The audience's interest in Ubuntu's recent decisions inspired me to tell CDMAG readers in detail that Ubuntu has always been eager to experiment.

For many users, the replacement of **GNU Coreutils** with **uutils** in Ubuntu was a major disappointment. Some viewed it as nothing more than *chasing hype* — after all, uutils is written in **Rust** and is being promoted in Ubuntu as part of the distro's supposedly stronger commitment to security.

But that's Ubuntu in a nutshell. *Twenty years ago*, this distro itself emerged to *fix* the long-standing issues in **Debian** — a claim that also sounded overly ambitious to many at the time. Since then, Ubuntu has become a testing ground for various ideas — some of them turned out to be huge successes, while others ended up as complete disasters. Here are a few examples of such decisions.

ShipIt

This was probably Ubuntu's most successful initiative — the one that not only popularized the distro itself but also opened a window into the Linux world for a much wider audience. Back in the days when the internet wasn't nearly as accessible as it is now, getting an ISO image of a system wasn't easy. Thanks to ShipIt, however, anyone could order a CD with the latest version of Ubuntu for free, while **Canonical** — the company behind Ubuntu — covered all the costs.

ShipIt lasted until **April 5, 2011** — that is, until it was no longer needed.

Upstart

Another successful Ubuntu project was its **init system** — one of the most critical components of any OS, responsible for booting and managing system services. The shortcomings of the classic **SysVinit** system motivated Ubuntu developers in **2006** to begin work on an alternative init system called **Upstart**. The strengths of **launchd** from Mac OS X (as it was then called, before being renamed to macOS) and **SMF** from Solaris served as major sources of inspiration and valuable ideas that helped shape the solution.

The new init system proved to be more efficient than SysVinit thanks to its *event-driven architecture*. System services could then be started and stopped based on events rather than runlevels. This approach enabled event generation on service start or stop, allowing other services to be bound to those events.

Upstart was even adopted by **Red Hat Enterprise Linux 6** and is still used in **ChromeOS** today — and will likely remain in use until the end of its days, when ChromeOS is gradually dissolving into **Android**.

systemd replaced Upstart in **Ubuntu 15.04**, and since then, the distro has shipped exclusively with systemd.

Unity, Mir, and Ubuntu Phone

Unifying the Ubuntu user experience across desktops and mobile devices was arguably Canonical's most ambitious initiative — a multi-project effort that took **seven years** to pursue.

In **2010**, Ubuntu's developers began rethinking the desktop as part of the **Unity** project, which at that time was essentially just a replacement for **GNOME Shell**. Back then, netbooks — those almost toy-sized laptops that were popular at the time, often with a SIM-card slot for mobile internet — were seen as a promising class of devices (now completely displaced by tablets with detachable keyboards). One of Unity's goals was to maximize available screen space, something the Ubuntu team aimed to achieve by moving away from GNOME Shell, whose design principles they disagreed with.

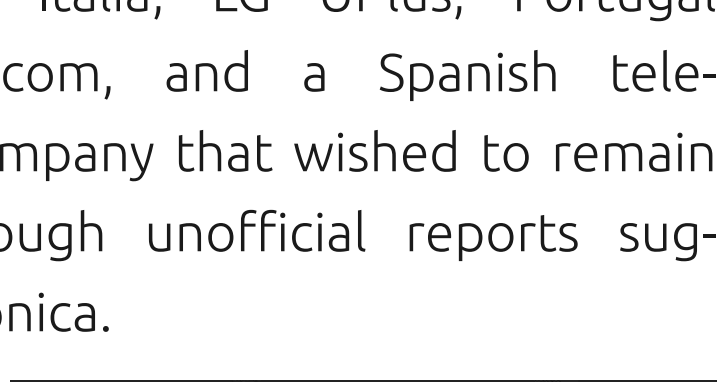
Ubuntu 11.04, released on **April 28, 2011**, shipped with Unity.

The year **2013** proved to be exceptionally eventful for Ubuntu: in just one year, Canonical went from being a desktop-oriented company to a full-fledged mobile platform vendor with **Ubuntu Phone**. Here are the key milestones from that year:

📆 In **January**, Canonical introduced the first edition of Ubuntu for smartphones, and by **February**, the first experimental builds of the mobile platform were already available.

📆 In **March**, Canonical announced that it was working on its *own graphics stack*, whose main components would be the Mir display server and the **Unity Next** desktop environment. Built with **Qt 5**, Unity Next no longer had anything in common with GNOME. It later became known as **Unity 8**, while the original Unity continued to be developed in parallel under the name **Unity 7**. At that point, it was still unclear why Ubuntu needed an entirely new graphics stack, but a few months later, new details emerged that shed light on the decision.

📆 In **June**, Canonical announced the creation of an independent advisory body, the **Carrier Advisory Group** (CAG). It intended to give mobile operators a voice in the development of Ubuntu for smartphones and to help shape the platform's strategy and direction. The founding members of the CAG included eight mobile operators: Deutsche Telekom, Everything Everywhere, Korea Telecom, Telecom Italia, LG UPlus, Portugal Telecom, SK Telecom, and a Spanish telecommunications company that wished to remain anonymous — though unofficial reports suggested it was Telefónica.



📆 Also in **June**, Canonical unveiled the **Ubuntu Edge** project — a smartphone Canonical hoped to fund through crowdfunding. The campaign set an extraordinarily ambitious goal: to raise **$32 million** in 30 days. This was three times higher than the crowdfunding record at the time, held by the Pebble smartwatch, which had raised about **$10 million**. Within the first four hours, the campaign brought in roughly $1 million, and by the end of the first day, it had reached $3.5 million.

The key feature of Ubuntu Edge was meant to be **convergence** — the ability to provide an adaptive mobile environment that, when connected to a monitor, would present a full desktop experience, effectively *turning a smartphone or tablet into a portable workstation*. This was precisely why the Mir display server and the Unity 8 desktop environment, announced in March, were needed.

Unfortunately, the crowdfunding campaign did not reach its ambitious goal, although it did set a new Kickstarter record by raising over **$12 million**. This marked the end of the Ubuntu Edge project, but it did not bury Ubuntu on smartphones. Canonical continued pursuing the goals of Ubuntu Edge through partnerships with various smartphone manufacturers.

📆 Also in **June**, one of Canonical's engineers working on Mir introduced an X11 compatibility layer for the display server, called **XMir**. He demonstrated running Xfce, LXDE, and GNOME 3 through it — the layer enabled the X.Org server to render its output via Mir. In practice, XMir was expected to be the main workhorse in the early stages, since, apart from Unity 8, other desktop environments and applications knew nothing about Mir at the time.

📆 **June** concluded with an announcement from **Oliver Ries**, the director of Mir and Unity at Canonical, who stated that **Ubuntu 13.10** would use Mir as the default display server, with XMir serving as the compatibility layer for running all popular desktop environments. At the time, it seemed as if the development of Mir and Unity 8 was moving at a rapid pace, and that both projects would reach stability within the next few months.

📆 But in **September**, the *first warning sign* appeared: Intel dropped support for Mir and XMir in its open-source video driver, **xf86-video-intel**. Initially, patches with experimental XMir support were included in xf86-video-intel **2.99.901**, but just a few days later, version **2.99.902** was released, with the patches removed and accompanied by a note stating that Intel did not *condone or support* Canonical's chosen course of action. As a result, Canonical was forced to maintain these patches on its own and ship a modified version of xf86-video-intel in Ubuntu.

📆 In **October**, the *second warning sign* followed: at the very last moment — after the final beta of Ubuntu 13.10 had already been published — Canonical reversed its decision to use Mir as the default display server. The XMir compatibility layer, critical for backward compatibility with desktop environments and applications that were still unaware of Mir, turned out not to be ready.

📆 Ubuntu for smartphones was not affected by this setback; therefore, in **October**, Canonical announced that Mir would be used there as the default display server. The mobile platform was being built from scratch with no regard for backward compatibility; therefore, legacy issues simply could not arise.

In **February 2015**, it was announced that the first smartphone running Ubuntu had gone on sale. The device was the **Aquaris E4.5 Ubuntu Edition**, priced at **€169.90**. The Ubuntu version shipped with the phone included a graphics stack based on Mir and Unity 8.

In **March 2014**, Canonical founder **Mark Shuttleworth** told the community at Ubuntu's virtual developer summit that Mir should not be expected to become the default display server in Ubuntu until the release of **Ubuntu 16.04**, scheduled for April 2016.

However, in **August 2015**, it became clear that Ubuntu 16.04 would continue using the old graphics stack based on Unity 7 and X.Org.

In **September 2015**, the *third warning sign* popped up: **Cristian Parrino**, the team lead for Ubuntu Phone, left Canonical.

In **May 2016**, Canonical once again cancelled its plan to adopt Mir as the default display server, and **Ubuntu 16.10**, like the previous release, continued to rely on the legacy graphics stack built on Unity 7 and X.Org.

Finally, on **April 5, 2017**, Mark posted an announcement on Canonical's blog declaring that the company was shutting down its efforts on Ubuntu for smartphones and Unity 8, and that Ubuntu would return to using GNOME. In that post, he wrote: *"I took the view that, if convergence was the future and we could deliver it as free software, that would be widely appreciated both in the free software community and in the technology industry [...] I was wrong on both counts"*.

Thus, after seven years of development, Canonical had invested a huge amount of money into Unity without ever figuring out how to make it commercially sustainable. The very next day, it became known that the company had been forced to lay off between **30%** and **60%** of its staff.

The Further Fate of the Projects

Ubuntu for smartphones is now known as **Ubuntu Touch** and is maintained by the non-profit **UBports** project. The platform continues to receive updates on a regular basis, and there is no sign of stagnation.

UBports also took over maintenance of Unity 8, renaming it to **Lomiri** in **July 2019** to avoid confusion with the Unity game engine. In **February 2023**, Lomiri was accepted into **Debian**.

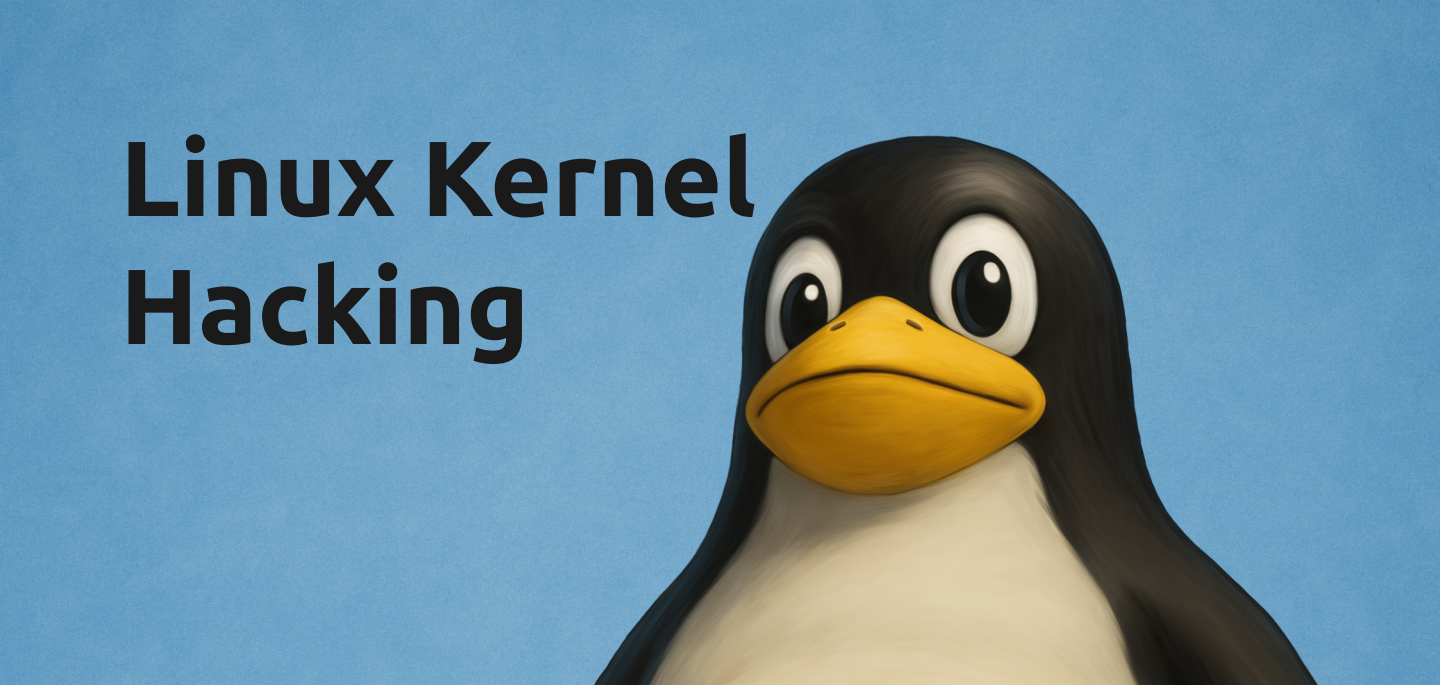
Mir continues to be developed by Canonical and plays an important role in the company's focus on IoT.

Conclusion

As you can see, Ubuntu has been through things far more dramatic than replacing GNU Coreutils with uutils. Some technical decisions worked out well — even inspired other distros — while others were ultimately unsuccessful. But no matter what the change was, it was always met with a wave of outrage from users and outside observers alike.

Keep in mind that **Ubuntu 25.10** is a "rehearsal" for the release of **Ubuntu 26.04**, which will be supported for **15 years**, so Canonical fully understands that everything has to work properly. I believe other distros will likely consider adopting uutils once the transition has been thoroughly validated in Ubuntu.

Linux Kernel Hacking



Fun fact: none of the Linux-kernel books I know explain a simple way to debug the kernel — whether it's a book focused on the kernel internals (Linux Kernel Development by Robert Love) or on writing modules (The Linux Kernel Module Programming Guide by Ori Pomerantz and Jim Huang). I recently bought two new Linux-kernel books (one from 2023 and another from 2024), skimmed them, and confirmed that the situation hasn't really improved over the years.

But at the same time, it's also quite sad, because it fuels the myth that kernel development is something extremely complex, accessible only to a chosen few. In reality, the kernel is not fundamentally different from any other large-scale software project. In fact, *in many cases*, developing and debugging the kernel and its modules isn't all that different from developing and debugging user-space applications.

User-Mode Linux Comes to the Rescue

User-Mode Linux (UML) is a virtualization system that allows Linux kernels to run as regular processes on a Linux host, which makes debugging dramatically easier. To achieve this, UML was implemented as a **port of Linux to Linux**.

The kernel has been ported to many hardware architectures so that it can run on devices ranging from microcontrollers to supercomputers, and UML can be viewed as a port to a software architecture.

Essentially, UML was the first virtualization system ever introduced in Linux — long before KVM and Xen appeared. UML has been around for over **20 years**: it officially became part of the kernel in **Linux 2.5.34**, released in **September 2002**.

Preparing an Empty Root Filesystem Image

To run a Linux kernel, we need two things: the *kernel* itself and a *root filesystem image* that contains a minimal environment based on **Debian**, **Ubuntu**, or **Alpine Linux**. Let's start by preparing the root filesystem.

You can think of a root filesystem image as a *virtual hard disk*. You've probably worked with hypervisors like VirtualBox, where one of the first steps is also preparing a virtual hard disk — typically in a more advanced format such as .vhd (Virtual Hard Disk).

Since we're working directly with an operating system kernel, we need to provide the bare minimum it requires. In this case, that's a virtual hard drive containing a basic Unix environment.

We'll create an empty 500 MB image. This should be enough for a basic environment, though you can increase the size if needed:

```
dd if=/dev/zero of=fs.img bs=1024 seek=$((
500 * 1024 )) count=1
```

In this command, dd skips 512k blocks of 1 KB each, then writes a single 1 KB block of zeros. The result is what is known as a *sparse file*. This approach allocates disk space only when data is actually written. So even though the image is nominally 500 MB, it takes up only the minimum required space on disk — typically one filesystem block (usually 4 KB). You can verify this:

```
du -sh fs.img
4.0K    fs.img
```

Now format the image. We'll use the **ext4** filesystem:

```
/sbin/mkfs.ext4 fs.img
```

Finally, mount the newly created filesystem, for example, at /mnt:

```
sudo mount -o loop fs.img /mnt
```

Populating the Root Filesystem Image

If you're using a **Debian**-based system (Linux Mint, Ubuntu, etc.), preparing the contents of the root filesystem is quite straightforward. If not, skip ahead to the universal method that follows — it works on any Linux system.

First, install debootstrap:

```
sudo apt update
sudo apt install debootstrap
```

Then build a **Debian 13** "Trixie" environment — the latest release of Debian GNU/Linux:

```
sudo debootstrap trixie /mnt https://ftp.de-
bian.org/debian
```

This command will populate `/mnt`, where your `fs.img` root filesystem image is mounted.

If you prefer Ubuntu, simply replace `trixie` with `numbat` to get an **Ubuntu 24.04** "Noble Numbat" environment.

Here's a universal, slightly more verbose method for creating an **Alpine Linux**-based environment:

```
ALPINE_VER=3.22
ARCH=x86_64
MIRROR=https://dl-cdn.alpinelinux.org/
alpine/
TOOLS_VER=$(curl -s "${MIRROR}/v${ALPINE_
VER}/main/${ARCH}/" | sed -n 's/.*apk-tools-
static-\(.*\)\.apk.*/\1/p')


wget "${MIRROR}/v${ALPINE_VER}/main/${ARCH}/
apk-tools-static-${TOOLS_VER}.apk"
tar -xzf apk-tools-static-"${TOOLS_VER}.apk"
sudo ./sbin/apk.static -X "${MIRROR}/
v${ALPINE_VER}/main" -U --allow-untrusted --
root /mnt --initdb add alpine-base
```

The last command will also populate `/mnt`, where the fs.img root filesystem image is mounted.

Adjust the variables as needed:

🔶 **Alpine Linux 3.22** was released in **May 2025** and is currently the latest version. If you need a different release, simply change the value of ALPINE_VER. The script above was written to be universal, so nothing should break.

🔶 If you want to build an environment for **ARM64**, set ARCH to `armhf`.

🔶 If you need a different mirror, you can pick one at https://mirrors.alpinelinux.org.

Once you're done preparing the root filesystem, do not forget to unmount it:

```
sudo umount /mnt
```

Building Linux

First, install the packages required to configure and build the Linux kernel.

On a **Debian**-based system, run:

```
sudo apt update
sudo apt install bc bison flex gcc libn-
curses-dev make wget xz-utils
```

On **Fedora** and **Red Hat Enterprise Linux** derivatives (AlmaLinux, Oracle Linux, Rocky Linux), run:

```
sudo dnf install bc bison diffutils flex gcc
make ncurses-devel wget xz
```

Next, download the archive with the latest kernel release, extract it, and start the configuration process:

```
wget https://cdn.kernel.org/pub/linux/ker-
nel/v6.x/linux-6.17.7.tar.xz
tar xJvf linux-6.17.7.tar.xz
cd linux-6.17.7
make ARCH=um menuconfig
```

Notice that in the last command, ARCH is set to um — this stands for User-Mode Linux, a port of Linux to Linux discussed at the beginning of the article.

You should now see the kernel configuration interface. For the moment, nothing needs to be enabled or disabled — just press **Exit** and confirm that you want to save the configuration. This will produce a .config file.

Now you can start building the kernel:

```
make ARCH=um -j4
```

Running Linux Under a Debugger

It's finally time to put everything together and run the Linux kernel under a debugger. Before we begin, let's take inventory: at the same directory level, you should have the `fs.img` file and the `linux-6.17.7` directory containing the kernel sources — and inside it, the compiled kernel binary named `linux`.

Once everything is in place, create a file named gdb_cmd next to `fs.img`:

```
set args ubd0=fs.img root=/dev/ubda rw handle
SIGSEGV pass nostop noprint init=/bin/dash -i
```

If you built an Alpine Linux environment, replace the value of `init` with `/bin/sh.`

Next, run:

```
gdb linux-6.17.7/linux -x gdb_cmd
```

When you see the debugger prompt (gdb), set a breakpoint on `start_kernel`, specify the directory containing the kernel sources, and then launch the kernel:

```
directory linux-6.17.7
b start_kernel
run
```

GDB may politely offer to download debug info for you:

```
This GDB supports auto-downloading debuginfo
from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or
[n])
```

Decline for now.

That's it — from this moment on, you're **officially** a kernel hacker: you've got the Linux kernel under a debugger, stopped at one of the very earliest stages of the boot process.

Time For the Simplest Module

Now you know how to debug the Linux kernel. In its current form, this is primarily helpful for understanding its internals: you read the source code, set a breakpoint where you're curious, and step through it line by line to see how a specific piece — or even an entire subsystem — works. But debugging is also extremely useful *when developing your own modules*. Remember how I said that developing the kernel and its modules isn't fundamentally different from developing any other large software project? That's absolutely true, which means that as kernel module developers, you need a proper debugger in your toolbox, not just print debugging (though we'll talk about that too).

Alright, let's dive in and see what **Hello, World!** looks like in the Linux kernel world.

Go to the Linux kernel source directory and create a `hello.c` module in `drivers/misc` with the following contents:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/printk.h>
static int __init hello_init(void)
{
    pr_info("Hello, World!\n");

    return 0;
}
static void __exit hello_exit(void)
{
    pr_info("Goodbye, World!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

Then add the following lines to the Kconfig file located in the same directory:

```
config HELLO
        tristate "Hello-World-style example
for a Linux kernel module"
        default n
        help
          This module is your first step
toward extending the Linux kernel.
```

Finally, add the following line to the Makefile located in the same directory:

```
obj-$(CONFIG_HELLO)             += hello.o
```

We will go over what all of this means soon, but for now, let's get our first working result as quickly as possible. To do that, go back to the root of the Linux kernel source tree and run the configurator again:

```
make ARCH=um menuconfig
```

In the menu that appears, find our module and set it to build as built-in.

```
Device Drivers ->
  Misc devices ->
    <*> Hello-World-style example for a Linux
kernel module"
```

Press the space bar on the line `Hello-World-style example for a Linux kernel module` until a * appears next to it. This means the module will be built as built-in. Built-in modules become part of the kernel itself, so you don't need to load or unload them separately — their code runs during kernel boot.

From here on, there's nothing new. Once you are done with the configuration, do not forget to tell the configurator to save it to the .config file. Then start the build:

```
make ARCH=um -j4
```

You will probably be surprised at how quickly the build completes. That's because most of the previous build is reused, and only the parts of the kernel that need to be built or rebuilt are processed.

After that, run the freshly built kernel under the debugger again:

```
gdb linux-6.17.7/linux -x gdb_cmd
```

When you enter the debugger, set the source directory just like before — but this time set a breakpoint on `hello_init`, the entry point of our module, and then start the kernel:

```
directory linux-6.17.7
b hello_init
run
```

The debugger should stop in the `hello_init` function.

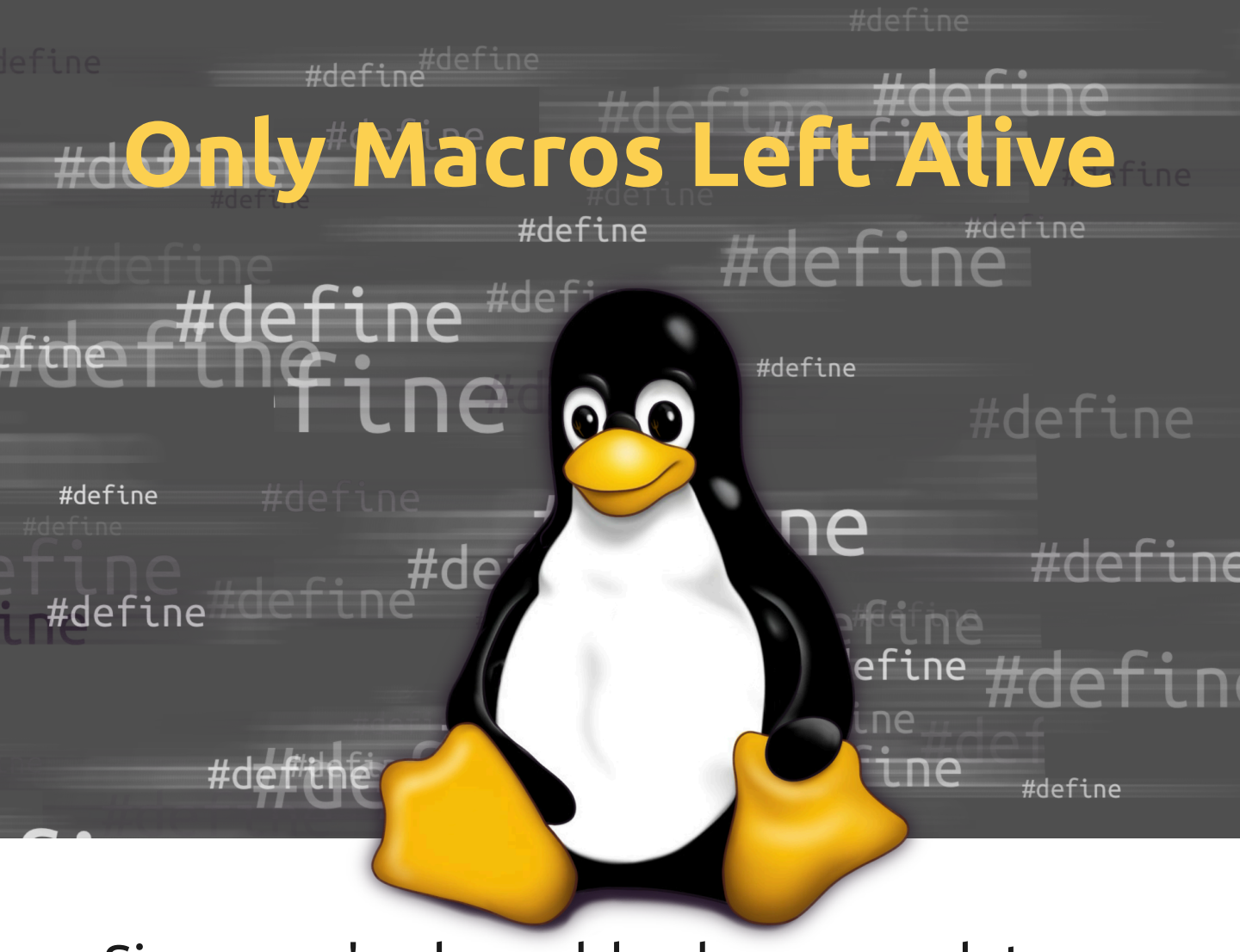
If you restart the debugger and run:

```
directory linux-6.17.7
run
```

Then, during the kernel boot, you should see **Hello, World**! You will also see the same message if you print the kernel logs using `dmesg`.

But why did we build the module as built-in? Simply to make debugging easier. Nothing stops you from distributing it separately later and letting users build it as a **loadable** module instead. But that already falls under the topic of module distribution.

Only Macros Left Alive



Since you're kernel hackers now, let me share some hacker wisdom — macros are the *longest-living creatures in the software world*. The Linux kernel has been evolving continuously for **34 years**, and you might think that none of the early code is still around and that it has been rewritten hundreds of times. But that's not the case. Let's take a look at one such example.

Here's a macro from **Linux 1.0**, released in **March 1994**. This macro is used to iterate over the entire list of processes.

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task)
!= &init_task ; )
```

And here's the same macro, but from **Linux 6.17**, released in **September 2025**.

```
#define for_each_process(p) \
    for (p = &init_task ; (p = next_task(p))
!= &init_task ; )
```

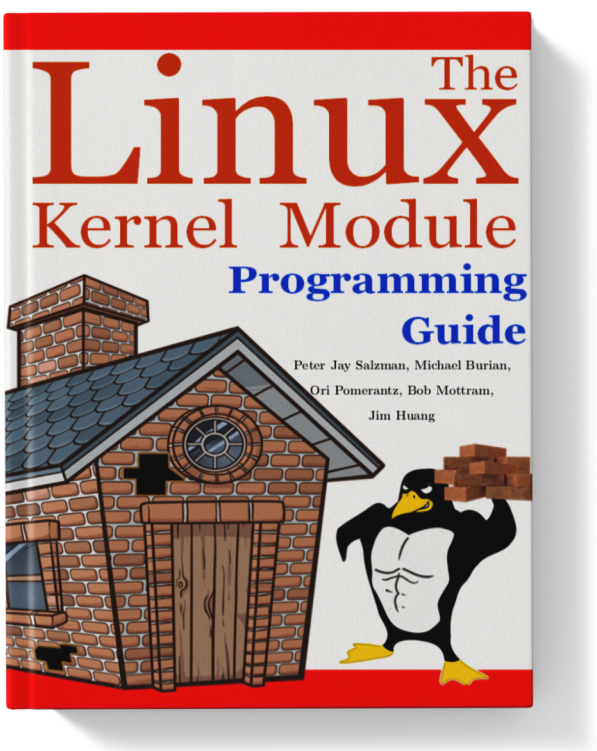
In a little over **30 years**, both the name and the way the next process is retrieved have changed: `for_each_task` became `for_each_process` when the kernel gained thread support, and the new macro now uses the **next_task** interface instead of accessing the process structure directly. Yes, things have changed — but at its core, it's still the same macro. CD

Q&A with Evgeny Golyshev

Q&A is a weekly series on the **CusDeb Magazine** Instagram page. Every **Friday**, I post a story inviting followers to ask their tech questions — and they always do. CDMAG you're reading now features the most interesting ones from **November**.

Q: Any tips on how to come up with a beginner-friendly Linux device driver project?

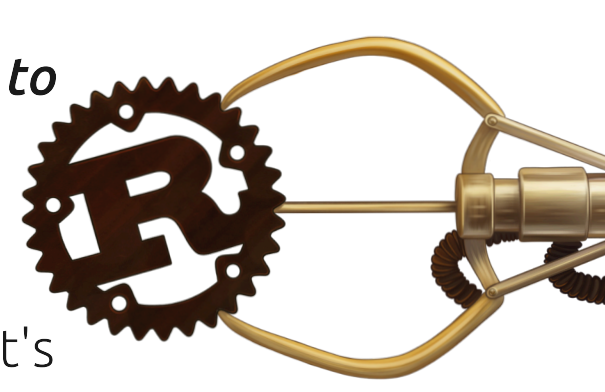
A: I recommend the book **The Linux Kernel Module Programming Guide**. It's free, and you can easily find it online. It includes a Hello-World-style example for a Linux kernel module. The only problem with books like this is that they don't explain how to run the Linux kernel under a debugger, which is essential when you want to develop and debug more complex modules.



Q: Let's say Linus decides to only support ARM64 in the Linux kernel. How much influence does the Linux Foundation board have on that decision? Can they reject the idea, or what would happen then?

A: The Linux Foundation can't influence Linus or control the development of Linux — that would violate his contract. His contract states that he can do whatever he wants as long as it remains open source. If he went crazy one day and decided to drop support for every architecture except ARM, no one would be able to stop him. But we could at least try calling his wife.

Q: Why is Debian switching to Rust?



A: Debian definitely stands out from other distros, but it's still developed in the same world — often by the same people who work on Ubuntu. In today's tech world, people love to blame everything on the lack of *memory safety*. I've recently posted a carousel post (sudo(-rs) vs. run0) explaining why that's not always true. There's nothing wrong with Rust — except maybe its syntax — and it's actually pushing other languages in the right direction.

Q: Do you think it's possible to make a living by contributing to open-source projects?

A: Yes, it's possible — just recently, there was a great example proving that. The Rust Foundation announced the "Maintainers Fund," an initiative to support the developers building Rust and the maintainers keeping the project alive. There are also plenty of companies developing commercial Linux systems, and they pay well for open-source work.

Team

Cover, illustrations, and layout: **Natalia Pisareva**

Editing: **Julia Babakova**

Jack-of-all-trades: **Evgeny Golyshev**

CusDeb Press