Online Chapter – Kernel Workspace Setup

This is an extension of *Chapter 1, Linux Kernel Programming - A Quick Introduction*, that has been published online. This chapter will teach you how exactly to setup the kernel workspace environment and get started.

We will install a recent Linux distribution, as a **Virtual Machine** (**VM**), and set it up to include all the required software packages. We will also clone this book's code repository on GitHub and learn about a few useful projects that will help along this journey.

Right at the outset, something I would like to emphasize is this: the best way to learn something is to do so *empirically, hands-on* – not taking anyone's word on anything at all but trying it out and experiencing it for yourself. Hence, this book gives you many hands-on experiments and kernel code examples that you can, and indeed must, try out yourself; this will greatly aid in your making real progress, learning deeply and understanding various aspects of Linux kernel and driver development.

Here's an indication of what you can expect to *do* as you progress through the book:

- Set up a working Linux kernel workspace on your system (as either a **VM** or a native system)
- Build the Linux kernel from source (for both the x86_64 as well as an ARM-based system)
- Understand the modern **Linux Kernel Module** (**LKM**) framework and leverage it to write kernel modules
- Write kernel modules (and a few user-space programs) that do various things (for example, iterate over all alive processes/threads, display user and kernel address-space details, dynamically manage kernel memory (allocating and freeing memory in various ways), and prod the kernel's out-of-memory killer, while also learning about CPU scheduling and cgroups through code examples)
- Understand the complexities inherent in working with concurrent (parallelized) hardware and software systems (like Linux) and learn when and how to synchronize your kernel code through various powerful technologies, both locking and lock-free.

So, always remember:

Be empirical! Also, be bold, be daring, and try things out!

Right, let's begin!

This chapter will take you through the following topics, which will help set up your working environment:

- Running Linux as a guest VM

- Installing an x86_64 Linux guest
- Additional useful projects

Technical requirements

You will need a modern – and preferably powerful – desktop PC or laptop. Ubuntu Desktop specifies "recommended minimum system requirements" for the installation and usage of the distribution here: `https://help.ubuntu.com/community/Installation/SystemRequirements`. I'd suggest you go with a system well beyond the minimum recommendation, as powerful a system as you can afford to use. This is because performing tasks such as building a Linux kernel from source is a very memory- and CPU-intensive process. It should be pretty obvious – the more RAM, CPU power, and disk space the host system has, the better!

Like any seasoned kernel developer, I would say that working on a native Linux system is best. However, for the purposes of this book, we cannot assume that you will always have a dedicated native Linux box available to you. So, we shall assume that you are working on a Linux guest. Working within a guest VM also adds an additional layer of isolation and thus safety. Of course, the downside is performance – working on a high-spec native Linux box can be up to twice as fast when compared to working on a VM!

Cloning our code repository: The complete source code for this book is freely available on GitHub at `https://github.com/PacktPublishing/Linux-Kernel-Programming_2E`. You can work on it by cloning the `git` tree, like so:

```
git clone https://github.com/PacktPublishing/Linux-Kernel-Programming_2E
```

The source code is organized chapter-wise. Each chapter is represented as a directory – for example, `ch1/` has the source code for this chapter. The root of the source tree has some code that is common to all chapters, such as the source files `convenient.h` and `klib.c`, as well as others.

For efficient code browsing, I would strongly recommend that you always index the code base(s) with `ctags` and/or `cscope`. For example, to set up the `ctags` index on a source tree, just `cd` to the root of the source tree and type `ctags -R`. (If you haven't already, please invest the time to learn code-browsing tools like cscope and ctags.)

Unless noted otherwise, the code output we show in the book is the output as seen on an x86_64 Ubuntu 22.04 LTS guest VM (running under Oracle VirtualBox 7.0). You should realize that due to (usually minor) distribution differences – and even minor differences within the same distributions but differing versions – the output shown in the book may not perfectly match what you see on your Linux system.

Running Linux as a guest VM

As discussed previously, a practical and often convenient alternative to using a native Linux system is to install and use the Linux distribution as a guest OS on a VM.

Selecting a Linux distro and kernel

It's key that you install a recent and well-supported Linux distribution, along with a recent long-term Linux kernel. In software, a **Long-Term Stable** (**LTS**) version, helps insulate you from constant maintenance and upgrades, and keeps your environment safe; the organization or people responsible for maintaining the LTS version will do so, keeping the product maintained and applying critical and required security and bug fixes, usually for a long-ish period. The scenario with respect to the LTS Linux kernel is mentioned shortly.

Very briefly, for this book, here's what we'll select:

- Linux distribution (or distro): Ubuntu 22.04 LTS (Jammy Jellyfish); free security and maintenance updates guaranteed until April 2027. EOL is April 2032.
- Linux kernel: The latest **LTS** Linux kernel version, as of the time of writing, **6.1.y**; **End of Life** (**EOL**) is December 2026 (the kernel version nomenclature is explained in detail in *Chapter 2*, *Building the 6.x Linux Kernel from Source - Part 1*).
- Hypervisor for the VM: Oracle VirtualBox 7.0.x (runs on the host system).

Our reasoning is simple: all of these are **Open-Source Software** (**OSS**) with, as of the time of writing, sufficiently long EOL dates, ensuring their continued support and viability for a long while.

You'll of course get all the details as we go along; for the Ubuntu VM installation details, the *Installing an x86_64 Linux guest* and *Setting up OSBoxes Ubuntu 22.04 as a guest OS* sections cover it. The Linux kernel versions, what exactly the nomenclature entails, and actually building a custom kernel are covered in depth in the following two chapters; relax, we'll get there.

Of course, running Linux on a native system has definite performance advantages. For the purpose of this book, though, we shan't assume you have a dedicated spare native Linux system, so we'll go with running Linux as a VM; it's also safer, helping avoid unpleasant data loss or other surprises. The fact is when working at the level of the kernel, abruptly crashing the system (and the data loss risks that arise thereof) is actually a commonplace occurrence.

> It's not just when working on kernel stuff that VMs are useful. In my personal experience, I've totaled a few Linux systems even when working on apps – all because of unfortunate (and quite silly) bugs! With a native system, this entails the painful re-installation and setup of the entire workspace, whereas with a VM, all you need to do is reinstall it (much quicker, typically) or work from an existing snapshot (even better). This is much easier, with less tension and headaches when things go wrong (remember Murphy's law? It does apply...). Of course, it's all a trade-off; with a native system, the performance can often be twice as high compared with a VM.

For the hypervisor, I recommend using Oracle VirtualBox 7.x (or the latest stable version) or other virtualization software, such as VMware Workstation.

Both of these hypervisors – Oracle VirtualBox and VMware Workstation – are freely available. It's just that the code for this book has been tested on VirtualBox 7.0.x. Oracle VirtualBox is considered OSS and is licensed under the GPL v2 (the same as the Linux kernel). You can download it from `https://www.virtualbox.org/wiki/Downloads`. Its documentation can be found here: `https://www.virtualbox.org/wiki/Documentation`.

The host system – the one where the hypervisor runs – should be a supported host: either MS Windows 10 or later (of course, even Windows 7 will work), a recent Linux distribution (for example, Ubuntu or Fedora), or macOS.

More choices

Ubuntu 22.04 LTS Desktop is the version of choice for this book. The two primary reasons for this are straightforward:

- Ubuntu Linux is one of the, if not *the*, most popular Linux development workstation environments in industry use today.
- We cannot always, due to a lack of space and clarity, show the code/build output of multiple environments in this book. Hence, we have typically chosen to show the output as seen on the x86_64 Ubuntu 22.04 LTS Desktop.

Ubuntu 20.04 – or even 18.04 LTS – Desktop is a good choice too (it has **Long-Term Support** (**LTS**) as well), and most things should work. To download it, visit `https://www.ubuntu.com/download/desktop`. But it is definitely older... (and this book's code hasn't been tested on them; a few issues, especially wrt kernel versions, are likely to show up).

Some other Linux distributions and/or hardware boards (along with their distro) that can also be considered include the following:

- **Fedora Workstation**: Fedora is a very well-known FOSS Linux distribution as well. You can think of it as being a kind of test-bed for projects and code that will eventually land within Red Hat's enterprise products. Download it from `https://getfedora.org/` (download the Fedora Workstation image; we do, at times, run some of this book's kernel code on x86_64 Fedora 38 and 39 VMs or native systems).
- **Raspberry Pi (ARM/ARM64) as a target**: It's really best to refer to the official documentation to set up your Raspberry Pi (*Raspberry Pi documentation*: `https://www.raspberrypi.org/documentation/`). It's perhaps worth noting that Raspberry Pi kits are widely available that come completely pre-installed and with some hardware accessories as well. We cover more on using the Raspberry Pi as a target in a later section.
- **BeagleBone Black (aka BBB; ARM) as a target**: The BBB, like the Raspberry Pi, is an extremely popular embedded ARM **Single-Board Computer** (**SBC**) for hobbyists and pros. You can get started here: `https://beagleboard.org/black`. The *System Reference Manual* for the BBB can be found here: `https://cdn.sparkfun.com/datasheets/Dev/Beagle/BBB_SRM_C.pdf`.

Though we typically don't present examples running on the BBB, nevertheless, it's a valid embedded Linux system that, once properly set up, you can run the majority of this book's code on.

- Of course, advanced readers will realize that the Linux system to use is really up to you. Running an as-light-as-possible custom Linux system on a QEMU (emulated) standard PC, using *Vagrant*, and so on are valid choices as well.

Before we conclude our discussion on selecting our software distribution for the book, here are a few more points to note:

- These distributions are, in their default form, FOSS and non-proprietary, and free to use as an end user.
- Though our aim is to be Linux distribution-neutral, the book's code has only been tested on an x86_64 guest running Ubuntu 22.04 LTS and lightly tested on the ARM-based (both ARM-32 and ARM-64) Raspberry Pi boards typically running a version similar to the Debian GNU/Linux OS. (On occasion, we also run our code on an (x86_64) Fedora 38/39 or Ubuntu 23.04 system.)
- We will, as much as is possible, use a recent (as of the time of writing) **stable LTS Linux kernel version 6.1** for our custom kernel builds and code runs. Being a long-term kernel with an EOL date of December 2026, the 6.1 kernel series is an excellent choice to run on and learn with.

It is interesting to know that, as of the time of writing, the 6.1 LTS kernels will indeed have a long lifespan, from December 2022 right up to (as of now) December 2026! This is good news: this book's content thus remains current and valid for years to come! Even better, the **Super LTS (SLTS) 6.1** kernel, maintained by the **Civil Infrastructure Platform** (**CIP**), will be maintained right up to August 2033! (You can find details on upcoming changes in the LTS model in the following chapter.)

It's important to realize that, for maximized security (with the latest defenses and fixes), you must strive to run the most recent stable long-term kernel possible for your project or product, and apply all updates, especially the security-related ones, as soon as they become available.

Now that we have chosen our Linux distribution, and/or hardware boards and VMs, it's time we install the guest along with setting up a user account and essential software packages.

So, let's get started with some basic information on installing our Linux guest (for the impatient, something to point out – an easier and quicker way to get started is to simply use pre-built Linux VM images! We show you how in the *Using pre-built Linux VM images* section).

Installing an x86_64 Linux guest

Here, I won't delve into the minutiae of installing Linux as a guest on Oracle VirtualBox, the reason being that this installation is *not* directly related to Linux kernel development. There are many ways to set up a Linux VM; we really don't want to get into the details and the pros and cons of each of them here.

But if you are not familiar with this, don't worry. For your convenience, here are some excellent resources that will help you out:

- From Ubuntu: *How to run an Ubuntu Desktop virtual machine using VirtualBox 7*: `https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox`
- A clearly written tutorial entitled *Install Linux Inside Windows Using VirtualBox* by Abhishek Prakash (*It's FOSS!, August 2019*): `https://itsfoss.com/install-linux-in-virtualbox/`
- An alternate, similarly excellent resource is *Install Ubuntu on Oracle VirtualBox*: `https://brb.nci.nih.gov/seqtools/installUbuntu.html`

Also, you can look up useful resources for installing a Linux guest on VirtualBox in the *Further reading* section for this chapter.

Nevertheless, while you install your x86_64 Linux VM, it's important to keep the following things in mind:

- Turn on your x86 system's virtualization extension support (within the system BIOS or UEFI).
- Allocate sufficient space to the guest disk; for most desktop/laptop systems, allocating 1 GB of RAM and two (or more) CPUs to the guest VM should be sufficient. However, when allocating space for the guest's disk, please be generous. Instead of the usual/default smaller amounts suggested, I strongly recommend you make it 50 GB or even more. Of course, this implies that the host system has more disk space than this available! Further, you can (preferably) specify this amount to be *dynamically allocated or allocated on-demand*. The hypervisor will "grow" the virtual disk optimally, not giving it the entire space to begin with. Allocating more disk space, as well as RAM, to the VM is good for its performance; furthermore, it allows one to increase disk/memory usage without having to worry that it will run out anytime soon. (A quick couple of tips: with respect to performance and storage, in VirtualBox, go to the **Settings dialog** | **Storage** | **Controller** and tick the **Use Host I/O Cache** box. Next, using the hypervisor's checkpoint-restore (or snapshots) features can be very useful but can result in a lot of (host) disk space getting used up; attaching a high-capacity external SSD for such purposes can certainly help).

Right, now let's get on with how you can more easily install a pre-built Linux as a guest.

Using pre-built Linux VM images

The **OSBoxes** (**OSB**) project allows you to freely download and use pre-built VirtualBox (as well as VMware) images for popular Linux distributions. See their site here: `https://www.osboxes.org/virtualbox-images/`.

In our case, we can download a prebuilt x86_64 Ubuntu 22.04 (as well as others) Linux image here: `https://www.osboxes.org/ubuntu/`. It comes with the VirtualBox Guest Additions (see the following info box for details) pre-installed!

What the heck are VirtualBox Guest Additions?

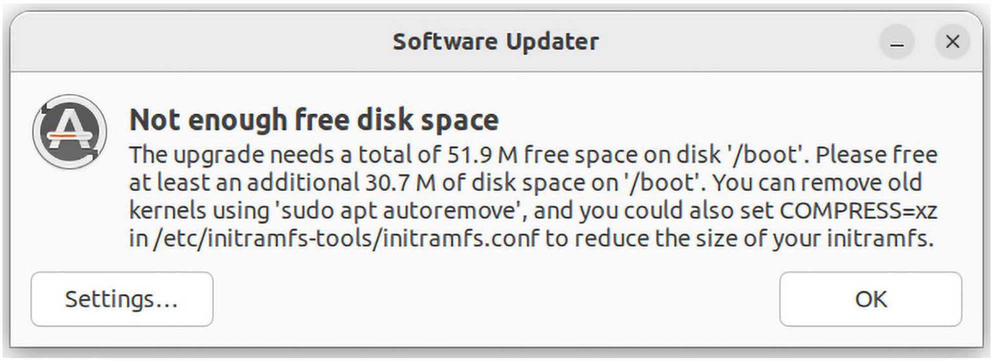
For best performance, it's important to install the Oracle VirtualBox Guest Additions as well within the guest VM. This is essentially para-virtualization accelerator software, which greatly helps with optimizing performance (especially regarding disk/network I/O, graphics, as well as providing USB3 support).

The default username/password is `osboxes/osboxes.org`.

In this book, I'll use the OSBoxes prebuilt Ubuntu 22.04 LTS as my primary VM. Its main advantage is that once downloaded, you're essentially good to go! Further, it comes with the VirtualBox Guest Additions virtual CD image.

There are a couple of downsides to OSB though:

- For a desktop-based development system, the boot partition is relatively small (about 250 MB); building many kernel images tends to fill this up quickly (you'll then have to delete older kernel images and their related artifacts from the boot partition mount point, typically `/boot`. This dialog (*Figure 14.1*) that I once saw when updating the system is typical of the issue; follow its advice:



Not enough free disk space

The upgrade needs a total of 51.9 M free space on disk '/boot'. Please free at least an additional 30.7 M of disk space on '/boot'. You can remove old kernels using 'sudo apt autoremove', and you could also set COMPRESS=xz in /etc/initramfs-tools/initramfs.conf to reduce the size of your initramfs.

Figure 14.1: A dialog popup showing that /boot is short on space! Follow its advice

- The default OSBoxes image is rather large; the 7-Zip-compressed image – the one to download – is about 2.6 GB in size; when uncompressed, it expands to approximately 8.6 GB (further, it's set up so that it can grow – storage is allocated dynamically, which is good).

Setting up OSBoxes Ubuntu 22.04 as a guest OS

Once you've downloaded (and uncompressed) the **OSB** Ubuntu 22.04 image file (it's typically named `Ubuntu 22.04 (64bit).7z`, or more recently, just `64bit.7z`), simply create a new guest OS within VirtualBox (by clicking on the **New** button), tweak it's **Settings** as required and desired, and try it out.

We've shown an example of doing this (refer to *Figure 14.2*), running the VirtualBox creation in Expert Mode:

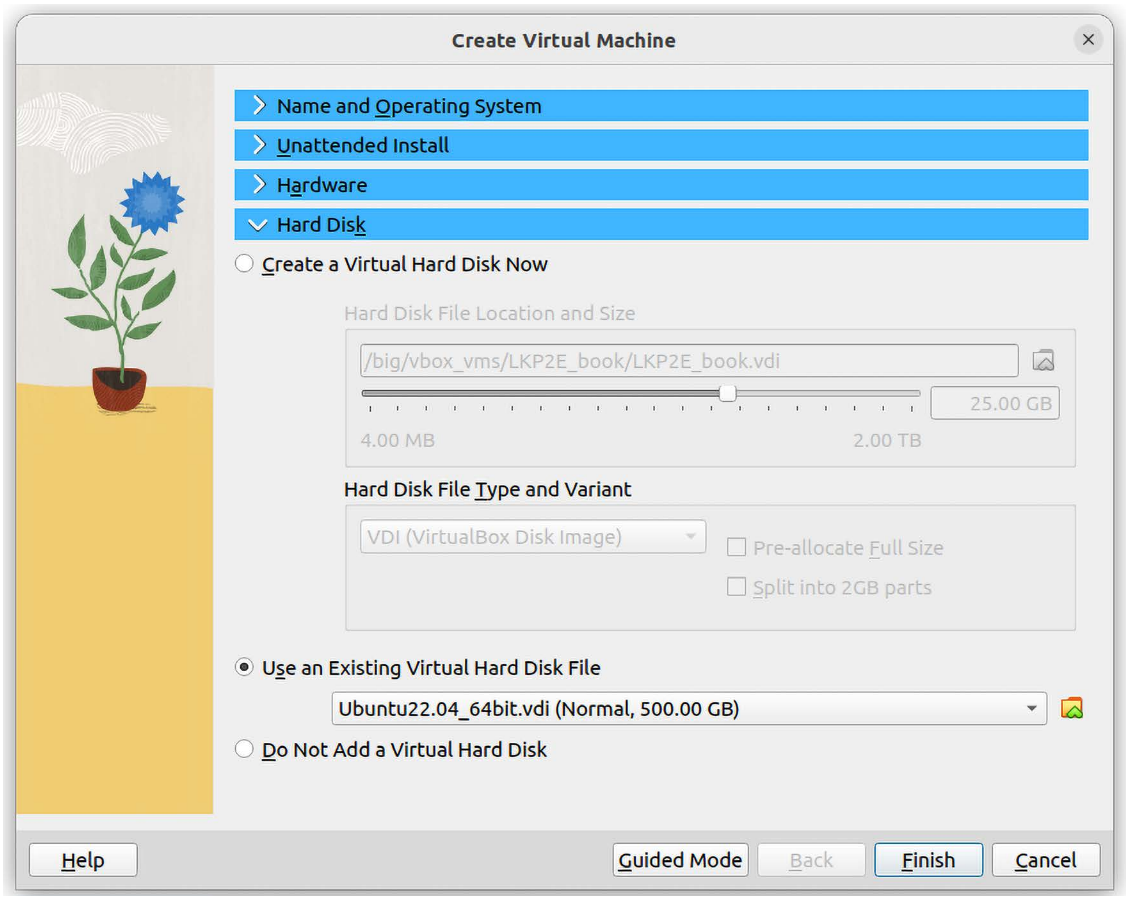

Create Virtual Machine
Name and Operating System
Unattended Install
Hardware
Hard Disk
Create a Virtual Hard Disk Now
Hard Disk File Location and Size
/big/vbox_vms/LKP2E_book/LKP2E_book.vdi
25.00 GB
4.00 MB
2.00 TB
Hard Disk File Type and Variant
VDI (VirtualBox Disk Image)
Pre-allocate Full Size
Split into 2GB parts
Use an Existing Virtual Hard Disk File
Ubuntu22.04_64bit.vdi (Normal, 500.00 GB)
Do Not Add a Virtual Hard Disk
Help
Guided Mode
Back
Finish
Cancel

Figure 14.2: Creating and setting up a new VirtualBox guest in Expert Mode

Especially when working with multiple images in VirtualBox, it's possible you'll get a well-known error, something along the lines of ... *Cannot register the hard disk <foo> <UUID> because a hard disk <bar> with <UUID> already exists....* To fix this, you'll need to assign a unique (internal) UUID (universal unique identifier) to the "new" virtual disk like this:

```
vboxmanage internalcommands sethduuid </path/to/new/virtual/disk.
vdi>
```

See details here: `https://poweradm.com/virtualbox-cannot-register-hard-disk-already-exists/`.

Okay, once you've ensured your new guest VM boots and it basically works, the tweaks we'll need to make to it are as follows:

1. Install the VirtualBox Guest Additions.
2. Set up a new user account named `c2kp`.
3. Install other required software packages.

Let's perform these steps! (Though these steps are shown with using the OSB Ubuntu 22.04 LTS, they should work for any Debian/Ubuntu based Linux guest.)

Step 1. Install the VirtualBox Guest Additions

1. Start up your OSB Ubuntu 22.04 VM (from within the VirtualBox app) on your host system.
2. Log in to your Linux guest as the user `osboxes` (recall that the default password is `osboxes.org`) within the (default) graphical environment.
3. A prerequisite to installing the Guest Additions is, we must first install some minimal packages; let's do so. In the Terminal app, type (it's possible you'll have to wait until an ongoing automated/unattended upgrade completes):

   ```
   sudo apt update
   sudo apt upgrade
   ```

4. Reboot the guest.
5. After logging in (again as the `osboxes` user), next ensure that these packages are installed:

   ```
   sudo apt install -y gcc make perl git build-essential dkms linux-headers-
   $(uname -r) ssh
   ```

   (This command should be typed on one line. The `-y` option switch has `apt` assume a `yes` answer to all prompts; careful though, this could be dangerous in other circumstances.)

6. From VirtualBox's menu, select **Devices** | **Insert Guest Additions CD image....** Now a "virtual" CD shows up. Note the pathname of its mount point (you can use the `df` command in the Terminal app to do so); on my system, it happens to be `/media/osboxes/Vbox_Gas_7.0.4`.
7. Now, within the Terminal application, do this:

   ```
   sudo /media/osboxes/Vbox_Gas_7.0.4/VBoxLinuxAdditions.run
   [...]
   ```

8. Follow the onscreen prompts. The VirtualBox Guest Additions (mostly kernel modules) are installed via this script; all you have to do once it's done is reboot the VM.

   Tip: Enabling the bidirectional clipboard (between host and guest – very useful). From the VirtualBox menu, select **Devices** | **Shared Clipboard** | **Bidirectional.**

9. On Oracle VirtualBox, to ensure that you have access to any shared folders you might have set up, you need to set the guest account to belong to the `vboxsf` group; you can do so like this (once done, you'll need to log in again, or perhaps even reboot, to have this take effect):

   ```
   sudo usermod -G vboxsf -a ${USER}
   ```

Good going, let's move on to the next step.

Step 2. Set up a new user account named c2kp

1. Okay, I'll assume you're logged in to the OSBoxes guest (as the user `osboxes`). Now setting up a new user account via the CLI is easy:

   ```
   sudo useradd -m c2kp -s /bin/bash
   sudo passwd c2kp
   ```

   Look up the man page on `useradd` to understand the options passed.

   This creates a new account named `c2kp`, along with the (skeleton, mostly empty) home directory, and sets up the password as well (please do provide a secure password). It automatically creates a group of the same name as well, which will be your primary group.

   On a Debian-based distro like this one, as the default account, `osboxes`, belongs to the `adm` and `sudo` groups (and `/etc/sudoers` allows it), you can exploit using the `sudo` command and thus run stuff as root (superuser); you simply have to enter your own password when prompted. Note, of course, that on production systems, stuff like this raises security concerns and will typically be constrained if not altogether disabled.

2. So, for our `c2kp` account to run stuff as root (to use `sudo`), which is important for us to be able to do, we need to make `c2kp` a member of the `adm` and `sudo` groups:

   ```
   sudo usermod -a -G adm,sudo c2kp
   ```

Let's verify it worked:

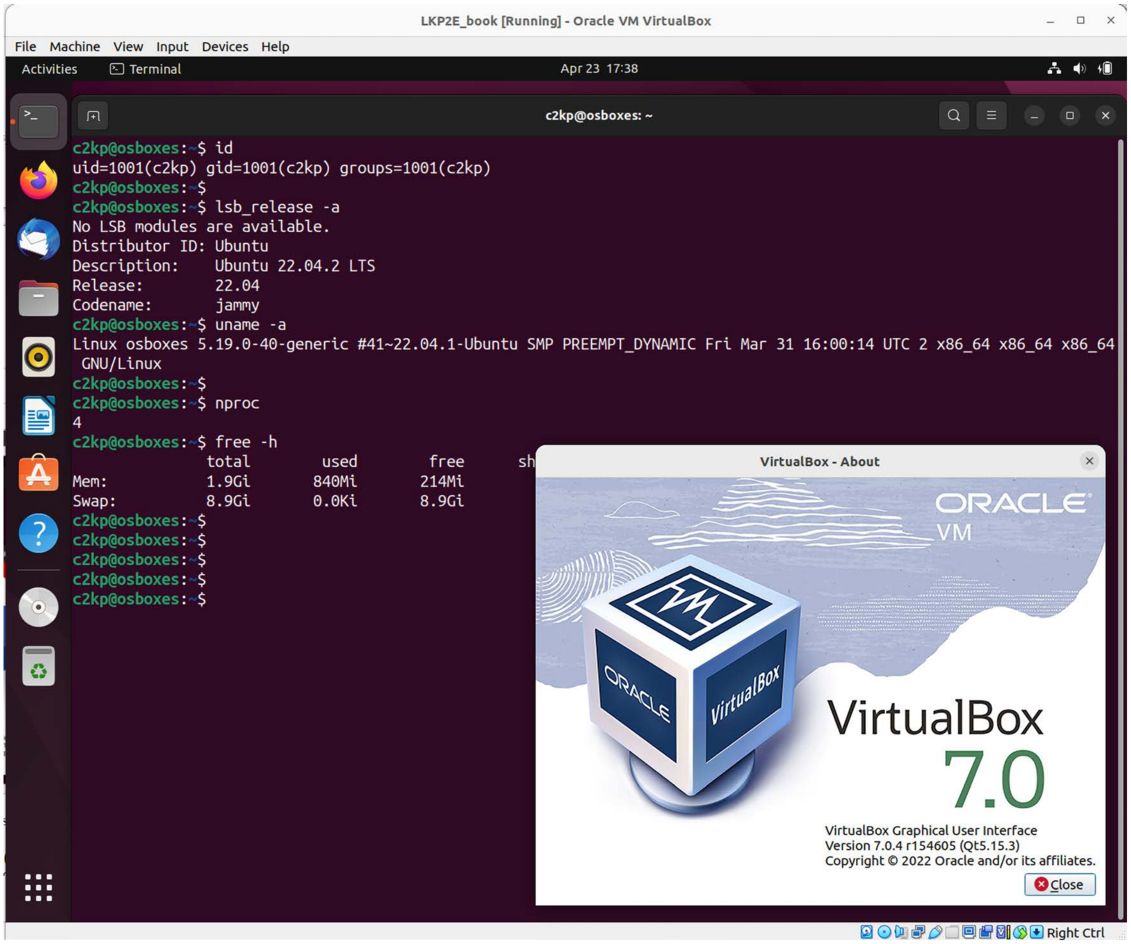
```
$ grep -E -w "adm|sudo" /etc/group
adm:x:4:syslog,osboxes,c2kp
sudo:x:27:osboxes,c2kp
```

Yes indeed! `c2kp` is now a member of the `adm` and `sudo` groups.

Wait, why on earth is our user account named `c2kp`?

Ah, thought you'd never ask. In the world of recreational running, there's a really well-known program to get people off their behinds; it's named *Couch to 5k*, abbreviated as *C25K* (`http://www.c25k.com/`). In a similar vein, we're using *Couch to Kernel Programmer*, or `c2kp`! (Hey, it's Linux, we prefer to keep account names in lowercase; also, take the name with a pinch of salt – I know you're smart!)

Great; log out and, from now onward, I'd suggest you always log in to the guest as the user `c2kp`. A screenshot of the VirtualBox About app in the foreground and our OSBoxes guest terminal window in the background follows:



```
c2kp@osboxes:~$ id
uid=1001(c2kp) gid=1001(c2kp) groups=1001(c2kp)
c2kp@osboxes:~$
c2kp@osboxes:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
c2kp@osboxes:~$ uname -a
Linux osboxes 5.19.0-40-generic #41~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Mar 31 16:00:14 UTC 2 x86_64 x86_64 x86_64
 GNU/Linux
c2kp@osboxes:~$
c2kp@osboxes:~$ nproc
4
c2kp@osboxes:~$ free -h
               total        used        free      sh
Mem:           1.9Gi       840Mi       214Mi
Swap:          8.9Gi       0.0Ki       8.9Gi
c2kp@osboxes:~$
c2kp@osboxes:~$
c2kp@osboxes:~$
c2kp@osboxes:~$
c2kp@osboxes:~$
```

VirtualBox Graphical User Interface
Version 7.0.4 r154605 (Qt5.15.3)
Copyright © 2022 Oracle and/or its affiliates.

Figure 14.3: Screenshot of the VirtualBox app and our freshly installed OSBoxes Ubuntu guest

You can see we're logged in as user `c2kp`. As my host system (it's also running Linux!) is quite powerful, I assign four CPU cores and 2 GB of RAM to the guest.

Let's now move on to a key step: actually installing the required software components on our Linux guest system so that, in the coming chapters, you can learn and write Linux kernel code on the system!

Step 3. Install required software packages

The packages that are installed by default when you use a typical Linux desktop distribution, such as any recent Ubuntu, Debian, CentOS, or Fedora Linux system, will likely include the minimal set required by a systems programmer: the native toolchain, which includes the GCC compiler along with headers, and the `make` utility/packages.

What is the toolchain and native toolchain?

To develop software, we (obviously) require a compiler (which compiles our source code to assembly and machine code so that it can execute on the machine). The venerable GCC compiler is typically the one we use (although *Clang* is becoming very popular!). But hang on, that's just the surface-level view; it's not just the compiler – looking deeper, there are many other tools required. They include the `make` utility, the `GCC compiler`, the standard (GNU) C library (glibc), the `binutils` package (the linker and assembler), the `bison` and `flex` parsers/lexical analyzers, the `GDB` debugger, and the GNU autotools suite. All these tools together make up what we call the **toolchain**, which is required for development. Furthermore, the toolchain that's installed by default on your typical Linux distro is called the *native toolchain* (it builds software from source code on that system for that same system type). On the other hand, we also have toolchains that build software on a host system but targeted to run on another (foreign) CPU architecture (like building on x86_64 for an ARM32); these are termed *cross-toolchains* (we shall come across them later in the book).

In this book, we are (also) going to learn how to write kernel-space code using an x86_64 VM and/or a target system running on a foreign processor (ARM32 and/or AArch64 being the typical cases). To effectively develop kernel code on these systems, we will need to install additional software packages.

Right, first of all, we'll assume you're running an x86_64 Ubuntu VM on VirtualBox with the Guest Additions installed. This is indeed the case when using the OSBoxes prebuilt Ubuntu image. (Just in case you still need to install them, no problem; refer to the section *Step 1. Install the VirtualBox Guest Additions*. Also, this tutorial has you covered: *How to Install VirtualBox Guest Additions in Ubuntu*: `https://www.tecmint.com/install-virtualbox-guest-additions-in-ubuntu/`.)

Next, let's cover the installation of some required packages.

This book comes with a code repo (on GitHub) here: `https://github.com/PacktPublishing/Linux-Kernel-Programming_2E`. We expect you to clone and use it as you progress through the book.

Hey, there's an easier way. I've provided **a Bash script in the book's GitHub repo that will install all required packages for Ubuntu Linux;** it can be found here: `https://github.com/PacktPublishing/Linux-Kernel-Programming_2E/blob/main/ch1/pkg_install4ubuntu_lkp.sh`. Do ensure you clone this book's code repository and then you can run it. Using our GitHub script also has the significant advantage that it will be kept updated...

To install the packages (manually), take the following steps:

1. Within the Ubuntu VM, first do the following:

```
sudo apt update
```

2. Now, to install the required packages for building the Linux kernel, run the following command on a single line:

```
sudo apt install -y \
 bison build-essential flex libncurses5-dev ncurses-dev \
libelf-dev libssl-dev tar util-linux xz-utils
```

3. To install the packages required for work we'll do in other parts of this book, run the following command:

```
sudo apt install -y \
  bc bpfcc-tools bsdextrautils \
  clang cppcheck cscope curl exuberant-ctags \
  fakeroot flawfinder \
  gnome-system-monitor gnuplot hwloc indent \
  libnuma-dev linux-headers-$(uname -r) linux-tools-$(uname -r) \
  man-db net-tools numactl openjdk-22-jdk  openssh-server \
  perf-tools-unstable psmisc python3-distutils  \
  rt-tests smem sparse stress sysfsutils \
  tldr-py trace-cmd tree tuna virt-what
```

Recall that the installation of `gcc`, `make`, and `perl` is done first (see the *Step 1. Install the VirtualBox Guest Additions* section) so that the Oracle VirtualBox Guest Additions can be properly installed straight after.

The disk space taken up by installing all these packages (at least on my VM) is in the region of 2.7 GB.

> This book, at times, mentions that running a program on another CPU architecture – typically AArch32 (ARM32) or AArch64 – might be a useful exercise. If you want to try (interesting!) stuff like this, I urge you to do so! Please read on; otherwise, feel free to skip ahead to the *Additional useful projects* section.

Do note that the book's GitHub repo does evolve; do a `git pull` occasionally to ensure you have the latest version of the code.

Installing QEMU and a cross-toolchain

One way to try things on an ARM machine is to actually do so on a physical ARM[64]-based SBC; for example, the Raspberry Pi is a very popular choice. In this case, the typical development workflow is to first build the ARM code on your x86_64 host system. But to do so, we need to install a **cross-toolchain** – a set of tools allowing you to build software on one host CPU such that it executes on a different (foreign) *target* CPU. An x86_64 *host* building programs for an ARM *target* is a very common case, and indeed is our use case here. Details on installing the cross-compiler follow shortly.

Often, an alternate way to just trying things out is to have an ARM/Linux system emulated – this alleviates the need for hardware! To do so, we recommend using the superb **QEMU** (**Quick Emulator**) project (`https://www.qemu.org/`).

To install the required QEMU packages, do the following (this takes up close to half a gigabyte of disk space on Ubuntu):

- For installation on Ubuntu, use the following:

  ```
  sudo apt install qemu-system-arm
  ```

- For installation on Fedora, use the following:

  ```
  sudo dnf install qemu-system-arm-<version#>
  ```

To get the version number on Fedora, just type the preceding command, and after typing the required package name (here, `qemu-system-arm-`), press the *Tab* key twice. It will auto-complete, providing a list of choices. Choose the latest version, and press *Enter*.

Installing a cross-compiler

If you intend to write a C (or C++) program that is compiled on a certain host system but must execute on another (foreign) target system, then you need to compile it with what's known as a cross-compiler or cross-toolchain. For example, in our use case, we want to work – develop code – on an x86_64 host machine. The host can even be an x86_64 *guest* system (that's fine), but the code must run on, say, an AArch32 (ARM32) target.

We shan't dig further into the specifics of installing a specific cross-toolchain here and now, as we practically require this – and explain it in depth – in a later chapter (*Chapter 3*, *Building the 6.x Linux Kernel from Source – Part 2*, in the *Step 2 – installing a cross-toolchain* section).

Another interesting thing to mention: you can use prebuilt (Docker) *containers* as cross-compile environments for multiple targets; it helps with guaranteeing that all teams on the project work with the identical cross-compile environment. Check out this page for more details (to learn and try): `https://github.com/dockcross/dockcross`.

A reasonable question you might have at this point is, why are we setting up a cross-toolchain in the first place? Although we aren't making use of it just now (after all, this is the "setup the workspace" material), we definitely shall later, in *Chapters 3* and *5*, and on a few other occasions, where you'll be configuring and cross-compiling a Linux kernel – and kernel modules – for the ARM[64] processor! Relax, we'll get there and dig into the details then.

Installing and using a cross-toolchain might require some reading up for newbie users. You can visit the *Further reading* section where I have placed a few useful links that will surely be of help.

A few remaining tips when running the VM

Sometimes, when the overhead of using the X Window System (or Wayland) GUI (for graphical display) is too high, especially on a guest machine, it's preferable to simply work in console mode. You can do so by appending `3` (the run level) to the kernel command line via the bootloader. However, working in console mode within VirtualBox may not be that pleasant an experience (for one, the clipboard is unavailable; two, the screen size and fonts are less than desirable).

Thus, doing a remote login (via `ssh`, `putty`, or equivalent) into the VM from the host system can be a great way to work (one that I use most of the time). This usually entails adding a "bridged mode" network adapter to the guest (shut down the guest, then in Oracle Virtualbox go to **Settings** | **Network**, and add it).

Next, remember to update the VM regularly and when prompted; this is an essential security requirement. (Otherwise, outdated vulnerable software continues to run on your system, making it a juicy target for malicious hackers who're always on the lookout for just this!) You can do so manually by using the following:

```
sudo /usr/bin/update-manager
```

Finally, to be safe, please do not keep any important data on the guest VM. We will be working on kernel development. Crashing the guest kernel can be pretty commonplace. While this usually does not cause data loss, you can never tell! To be safe, always back up any important data.

A tip: You can exploit VirtualBox's "Snapshots" feature, allowing you to keep, in effect, a known working point, and be able to restore it on demand.

Experimenting with the Raspberry Pi

The Raspberry Pi is a very popular credit card-sized (or smaller, as with the Raspberry Pi Zero boards) SBC, much like a small-factor PC that has USB ports, a microSD card, HDMI, audio, Ethernet, GPIO, and more. It's used for learning, by hobbyists, prototyping, and several real-world products as well. The **System on Chip** (**SoC**) that powers it is from Broadcom, and in it is an ARM core or cluster of cores.

Though not mandatory, in this book we strive to also test and run our code on some of these embedded systems (the Raspberry Pi Zero W and the Raspberry Pi 4 Model B boards). Running your code on different target architectures is always a good eye-opener to possible defects (bugs), and helps with testing and learning. I encourage you to do the same.

So what does the Raspberry Pi SBC look like? *Figure 14.4* shows you a typical specimen, a Raspberry Pi 4 Model B:

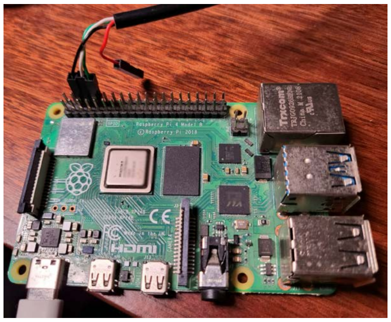


Figure 14.4: The Raspberry Pi 4 Model B with a USB-to-RS232 TTL UART serial adapter cable attached to its GPIO pins

You can work on the Raspberry Pi target either using a digital monitor/TV via HDMI as the output device and a traditional keyboard/mouse over its USB ports or, more commonly for developers, over a remote shell via `ssh`. The SSH approach, though, does not cut it in all circumstances. Having a *serial console* on the Raspberry Pi helps, especially when doing stuff like kernel (or driver) bring-up and debugging.

I recommend that you check out the following article, which will help you set up a USB-to-serial connection, thus getting a console login to the Raspberry Pi from a PC/laptop: *WORKING ON THE CONSOLE WITH THE RASPBERRY PI*, kaiwanTECH: `https://kaiwantech.wordpress.com/2018/12/16/working-on-the-console-with-the-raspberry-pi/`.

Of course, my *Linux Kernel Debugging* book (Packt Publishing, August 2022), covers kernel debugging tools and techniques in depth; do check it out.

To set up your Raspberry Pi, please refer to the official documentation: `https://www.raspberrypi.org/documentation/`. As of the time of writing, my Raspberry Pi system runs the "official" Debian Linux for Raspberry Pi; it's called the Raspberry Pi OS (it used to be called Raspbian) and sports a very recent 6.1-based Linux kernel. (Later, in the following two chapters, you'll not only learn how to build your own custom Linux kernel, but also one specifically for the Raspberry Pi!)

On the console (or a Terminal window) of the Raspberry Pi, to look up version details, we run the following commands:

```
rpi $ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 11 (bullseye)
Release:        11
Codename:       bullseye
rpi $ uname –a
Linux rpi 6.1.21-v8+ #1642 SMP PREEMPT Mon Apr  3 17:24:16 BST 2023 aarch64
GNU/Linux
rpi $
```

Quick tip

There are several interesting commands to query hardware/software info; try `hostnamectl`, `ls{cpu|pci|usb}`. Also, specific to x86, try `hwinfo`, `lshw`, and specific to the Raspberry Pi OS, the `raspinfo` commands, for even more details.

Also, of course, there is no reason to confine yourself to the Raspberry Pi family; there are several other excellent prototyping/evaluation boards available. One that springs to mind is the popular **BeagleBone Black** board.

In fact, for professional development and product work, the Raspberry Pi is perhaps not the best choice, for several reasons… a bit of googling will help you understand this. Having said that, as a learning and prototyping environment, it's hard to beat, with the strong community (and tech hobbyist) support it enjoys.

Several modern choices of microprocessors for embedded Linux (and much more) are curated, discussed, and contrasted in this excellent in-depth article: *SO YOU WANT TO BUILD AN EMBEDDED LINUX SYSTEM?*, Jay Carlson, October 2020: `https://jaycarlson.net/embedded-linux/`; do check it out.

By now, I expect that you have set up Linux as a guest machine (or are using a native "test" Linux box) and have cloned the book's GitHub code repository. So far, we have covered some information regarding setting up Linux as a guest (as well as optionally using boards such as the Raspberry Pi or the BeagleBone).

If you do face issues with setup, or later with code (of course, it happens to the best of us), remember, most answers are only a "google" away! Still, there are times when a well-thought-out question posted to relevant forums fetches a better human-generated answer. And then these days there are the generative AI LLMs, of course… again, **YMMV** (**Your Mileage May Vary**). As an additional help, we (the publisher and I) do try and support folks who write in their issues, using the book's GitHub repo as the main means of doing so (raise an issue, or PR (pull request), or simply write in).

So, congratulations! This completes the software setup. Now, let's check out a few additional and useful projects to complete this chapter. It's certainly recommended that you read through this section as well.

Additional useful projects

This section brings you details of some additional miscellaneous projects that you might find very useful indeed. In a few appropriate places in this book, we refer to or directly make use of some of them, thus making them important to understand.

Let's get started with the well-known and important Linux *man pages* project.

Using the Linux man pages

You must have noticed the convention followed in most Linux/Unix literature:

- The suffixing of *user commands* with `(1)` – for example, `gcc(1)` or `gcc.1`
- *System calls* with `(2)` – for example, `fork(2)` or `fork().2`
- *Library APIs* with `(3)` – for example, `pthread_create(3)` or `pthread_create().3`

The number in parentheses (or after the period) denotes the section of the **manual** (the **man** pages) that the command/API in question belongs to. A quick check with `man(1)`, via the `man man` command (that's why we love Unix/Linux!), reveals the sections of the Unix/Linux manual:

```
$ man man
[...]
```

```
A section, if provided, will direct man to look only in that section of the
manual. [...]

       The table below shows the section numbers of the manual followed by the
types of pages they contain.

       1   Executable programs or shell commands
       2   System calls (functions provided by the kernel)
       3   Library calls (functions within program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions eg /etc/passwd
       6   Games
       7   Miscellaneous (including macro packages and conventions), e.g.
man(7), groff(7)
       8   System administration commands (usually only for root)
       9   Kernel routines [Non standard]
[...]
```

So, for example, to look up the man page on the `stat(2)` system call, you would use the following:

```
man 2 stat           # (or: man stat.2)
```

At times though, the `man` pages are simply too detailed to warrant reading through when a quick answer is all that's required. Enter the `tldr` project – read on!

The tldr variant

While we're discussing `man` pages, a common annoyance is that the `man` page on a command is, at times, simply too large. Take the `ps(1)` utility as an example. It has a really large `man` page as, of course, it has a huge number of option switches. Wouldn't it be nice, though, to have a simplified and summarized "common usage" page for ps? This is precisely what the `tldr` pages project aims to do.

TL;DR means Too Long; Didn't Read.

In the tldr project's own words: "*The tldr pages are a community effort to simplify the beloved man pages with practical examples*" (I love the wording – the *beloved* man pages indeed!).

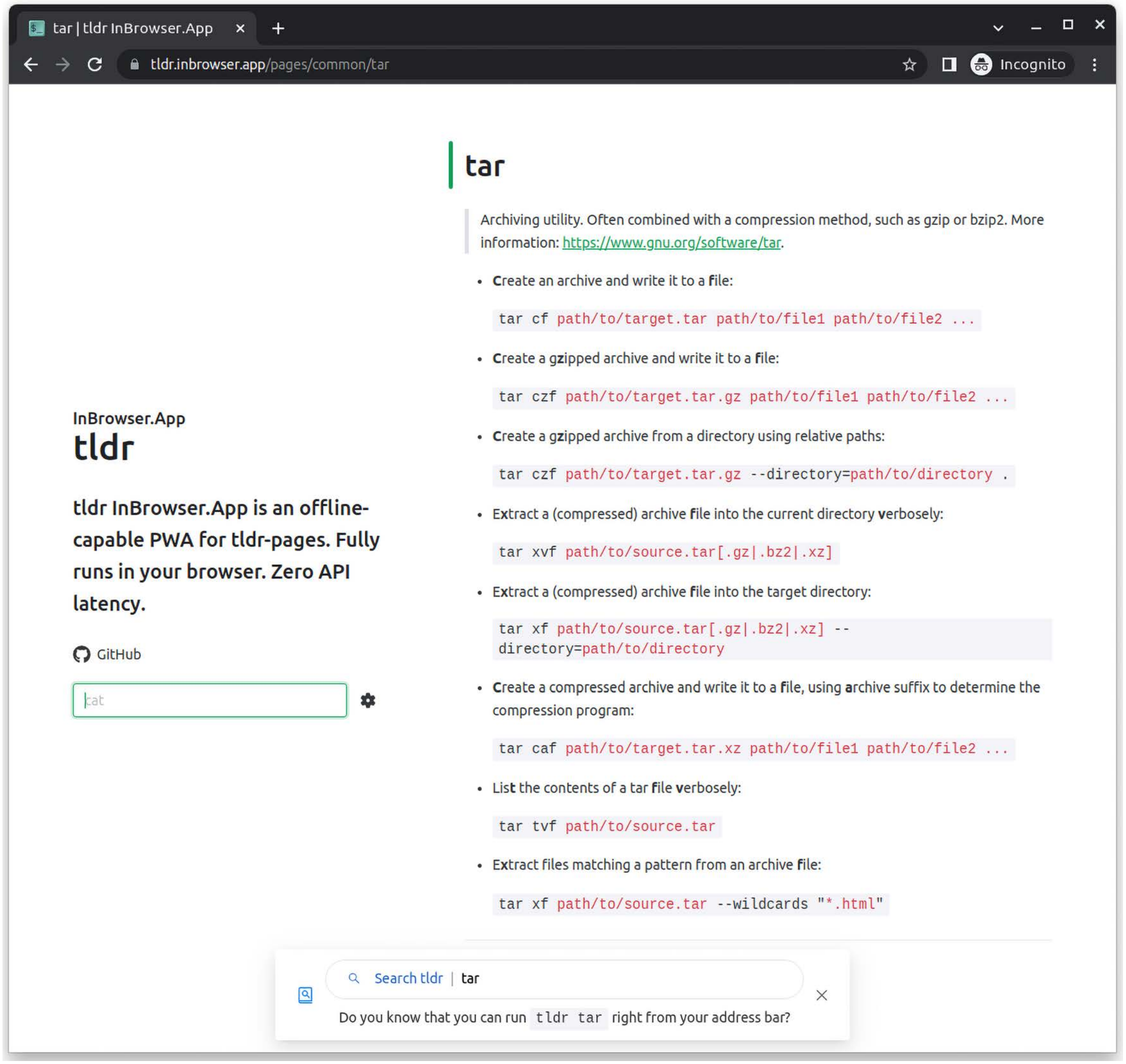
The tldr project seems to be pretty popular, with a large number of spin-offs, implementing the same idea in different ways. Check these out:

- The main website: `https://tldr.sh/`
- The tldr wiki site with various clients: `https://github.com/tldr-pages/tldr/wiki/tldr-pages-clients`

Using the tldr web client

Practically, let's use it via this web client `https://tldr.inbrowser.app/`. Head over to it, simply type in the command name (I used `tar`), and see the result instantly pop up (*Figure 14.5*).

Very interesting, the URL now becomes `https://tldr.inbrowser.app/pages/common/tar`. Well, a picture's worth a thousand words:



tldr InBrowser.App is an offline-capable PWA for tldr-pages. Fully runs in your browser. Zero API latency.

tar

Archiving utility. Often combined with a compression method, such as gzip or bzip2. More information: https://www.gnu.org/software/tar.

- Create an archive and write it to a file:

```
tar cf path/to/target.tar path/to/file1 path/to/file2 ...
```

- Create a gzipped archive and write it to a file:

```
tar czf path/to/target.tar.gz path/to/file1 path/to/file2 ...
```

- Create a gzipped archive from a directory using relative paths:

```
tar czf path/to/target.tar.gz --directory=path/to/directory .
```

- Extract a (compressed) archive file into the current directory verbosely:

```
tar xvf path/to/source.tar[.gz|.bz2|.xz]
```

- Extract a (compressed) archive file into the target directory:

```
tar xf path/to/source.tar[.gz|.bz2|.xz] --
directory=path/to/directory
```

- Create a compressed archive and write it to a file, using archive suffix to determine the compression program:

```
tar caf path/to/target.tar.xz path/to/file1 path/to/file2 ...
```

- List the contents of a tar file verbosely:

```
tar tvf path/to/source.tar
```

- Extract files matching a pattern from an archive file:

```
tar xf path/to/source.tar --wildcards "*.html"
```

Search tldr | tar

Do you know that you can run `tldr tar` right from your address bar?

Figure 14.5: A screenshot of one of the tldr web clients in action

Moving along, you'll recall that, with respect to APIs, we said that user space system calls fall under section 2 of the man pages, library subroutines under section 3, and kernel APIs under section 9. Given this, then, in this book, why don't we specify the, say, `printk` kernel function (or API) as `printk(9)` – as `man man` shows us that section 9 of the manual is *Kernel routines*?

Well, it's fiction, (at least on today's Linux): *no man pages actually exist for kernel APIs!* So, how do you get documentation on the kernel APIs, development model, and so on? That's just what we will briefly delve into in the following section.

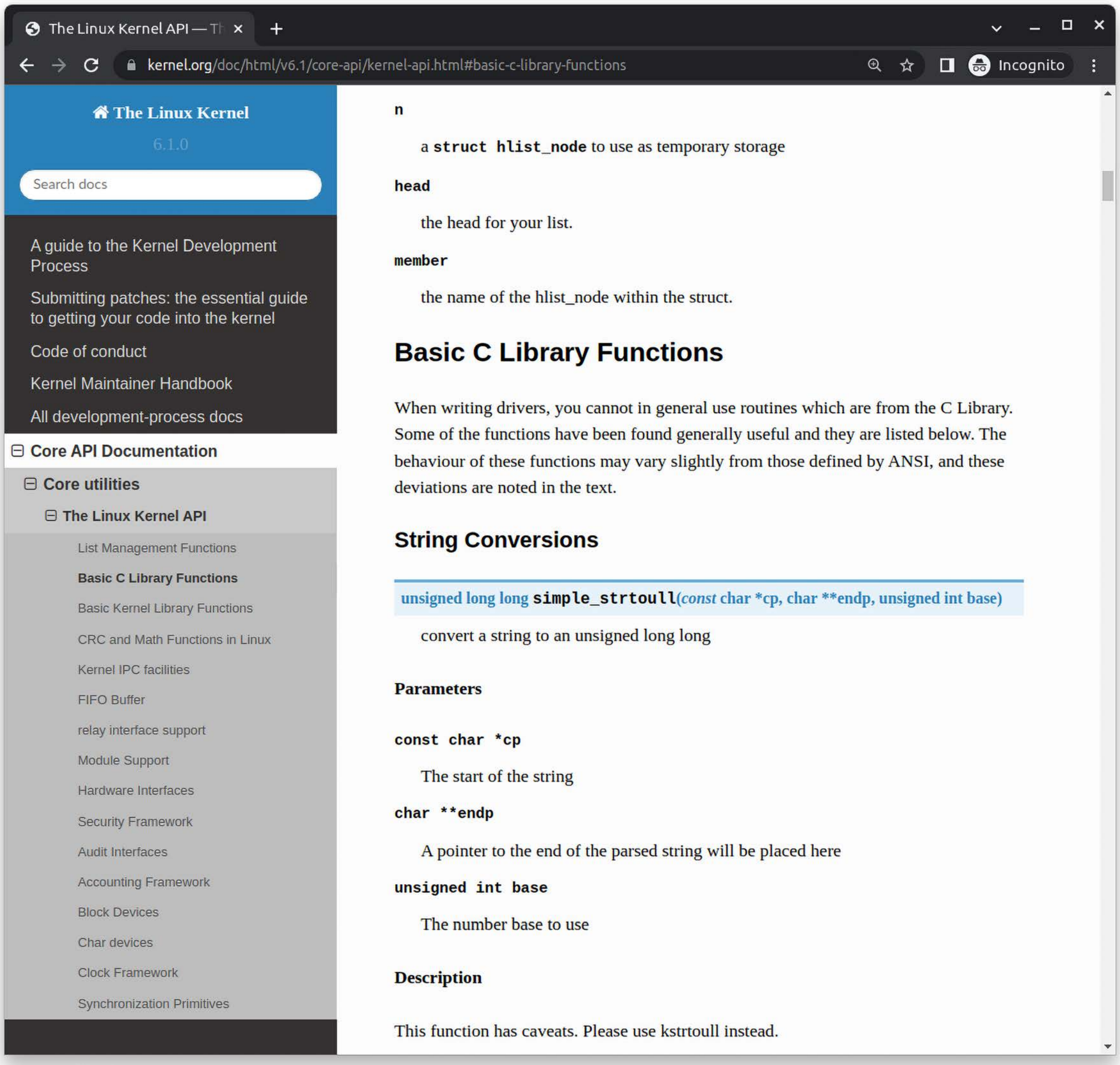
Locating and using the Linux kernel documentation

The Linux kernel isn't a toy project; it's an industry-strength, solid, proven OS, an engineer's OS that's all about performance in even very demanding situations (within it's design envelope). Given this, internal design and architecture decisions do tend to get complex; understanding them is critical. The elaborate kernel documentation comes to the rescue! It documents in detail not just the APIs and such, but also, more importantly, major design decisions.

The community has developed and evolved the Linux kernel documentation into a good state over many years of effort. The *latest version* of the kernel documentation, presented in a nice and modern web style, can always be accessed online here: `https://www.kernel.org/doc/html/latest/`.

(Of course, as we will mention in the next chapter, the kernel documentation is always available for that kernel version within the kernel source tree itself, within the directory named `Documentation/`.)

It's always advisable to look up the documentation for the kernel version you're working on. As just one example of the online kernel documentation, for the kernel we'll be using (6.1 LTS), see the following partial screenshot of the page on *Core Kernel Documentation/Basic C Library Functions* (`https://www.kernel.org/doc/html/v6.1/core-api/kernel-api.html#basic-c-library-functions`):



The Linux Kernel API — Th
kernel.org/doc/html/v6.1/core-api/kernel-api.html#basic-c-library-functions
Incognito
The Linux Kernel
6.1.0
Search docs
A guide to the Kernel Development Process
Submitting patches: the essential guide to getting your code into the kernel
Code of conduct
Kernel Maintainer Handbook
All development-process docs
Core API Documentation
Core utilities
The Linux Kernel API
List Management Functions
Basic C Library Functions
Basic Kernel Library Functions
CRC and Math Functions in Linux
Kernel IPC facilities
FIFO Buffer
relay interface support
Module Support
Hardware Interfaces
Security Framework
Audit Interfaces
Accounting Framework
Block Devices
Char devices
Clock Framework
Synchronization Primitives

`n`

a `struct hlist_node` to use as temporary storage

`head`

the head for your list.

`member`

the name of the hlist_node within the struct.

Basic C Library Functions

When writing drivers, you cannot in general use routines which are from the C Library. Some of the functions have been found generally useful and they are listed below. The behaviour of these functions may vary slightly from those defined by ANSI, and these deviations are noted in the text.

String Conversions

unsigned long long `simple_strtoull`(*const* char *cp, char **endp, unsigned int base)

convert a string to an unsigned long long

Parameters

`const char *cp`

The start of the string

`char **endp`

A pointer to the end of the parsed string will be placed here

`unsigned int base`

The number base to use

Description

This function has caveats. Please use kstrtoull instead.

Figure 14.6: Partial screenshot showing a small part of the modern online Linux kernel (v6.1) documentation

As can be gleaned from the screenshot, the modern documentation is pretty comprehensive. It even advises you that the `simple_strtoull()` API is considered obsolete, and to use the `kstrtoull()` API instead (see the last line in *Figure 14.6*).

Generating the kernel documentation from source

You can literally generate the full Linux kernel documentation from within the kernel source tree in various popular formats (including PDF, HTML, LaTeX, EPUB, or XML), in a *Javadoc* or *Doxygen-like* style. The modern documentation system used internally by the kernel is called **Sphinx**. Using `make help` within the kernel source tree will reveal several *documentation targets*, among them `htmldocs`, `pdfdocs`, and more. So, you can, for example, `cd` to the kernel source tree and run `make pdfdocs` to build the complete Linux kernel documentation as PDF documents (the PDFs, as well as some other meta-docs, will be placed in `Documentation/output/latex`). The first time, at least, you will likely be prompted to install several packages and utilities (we don't show this explicitly).

Don't worry if the preceding details are not crystal clear yet. I suggest you first read *Chapter 2, Building the 6.x Linux Kernel from Source – Part 1*, and *Chapter 3, Building the 6.x Linux Kernel from Source – Part 2*, and then revisit these details.

Static analysis tools for the Linux kernel

Static analyzers are tools that, by examining the source code, attempt to identify potential errors, and even security-related issues, within it. They can be tremendously useful to you as the developer, though you must learn how to "tame" them – in the sense that they can often generate false positives.

Several useful static analysis tools exist. Among them, the ones that are more relevant for Linux kernel code analysis include the following:

- Sparse: `https://sparse.wiki.kernel.org/index.php/Main_Page`
- Coccinelle: `http://coccinelle.lip6.fr/` (requires the `ocaml` package installed)
- Smatch: `http://smatch.sourceforge.net/` and `http://repo.or.cz/w/smatch.git`

In addition, these are general-purpose C/C++ static analyzers that can be useful as well:

- Flawfinder (simple, geared toward finding security issues): `https://dwheeler.com/flawfinder/`
- Cppcheck: `https://github.com/danmar/cppcheck`

Of course, there are also several high-quality commercial static analysis tools available. Among them are the following:

- SonarQube: `https://www.sonarqube.org/` (a free and open-source community edition is available)
- Coverity Scan: `https://scan.coverity.com/`
- Klocwork: `https://www.meteonic.com/klocwork`

Clang is a compiler frontend to GCC that is becoming popular even for kernel builds; it has a static analysis component as well. (Clang is in fact the compiler used to build Android.)

Static analysis tools can save the day. Time spent learning how to use them effectively is time well spent!

LTTng – the Linux Trace Toolkit next generation

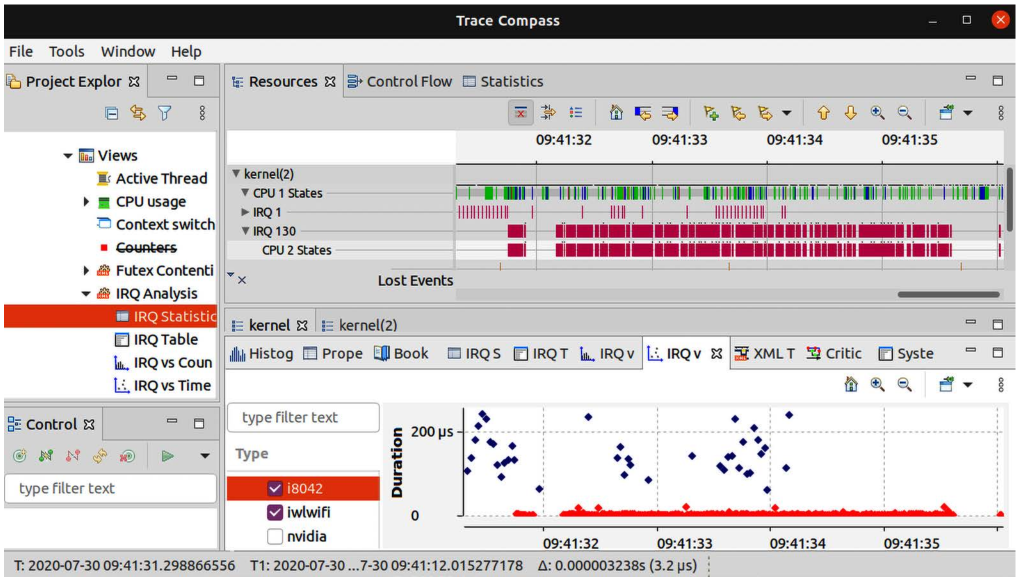
At times, often during performance analysis, and even for debugging, it's key to understand what code paths are taken (most often), and where bottlenecks lie. Profiling and tracing tools often come to the rescue in such cases. A superb tool for *tracing* and *profiling* is the powerful **Linux Tracing Toolkit next generation** (**LTTng**) toolset, a Linux Foundation project. LTTng allows you to trace both user space (applications) and/or the kernel code paths in minute detail. This can tremendously aid you in understanding where performance bottlenecks occur, as well as aiding you in understanding the overall code flow and thus in debugging scenarios, learning about how the code actually performs its tasks.

In order to learn how to install and use it, I refer you to its very good documentation here: `https://lttng.org/docs` (try `https://lttng.org/download/` for installation for common Linux distributions).

It's also highly recommended that you install the **Trace Compass GUI**: `https://www.eclipse.org/tracecompass/`. It provides an excellent GUI for examining and interpreting LTTng's output.

Trace Compass minimally requires a **Java Runtime Environment** (**JRE**) to be installed as well. We've installed one on our Ubuntu 23.04 LTS system – the `openjdk-22-jdk[-headless]` package.

As an example (I can't resist!), here's a screenshot of a capture by LTTng being "visualized" by the superb Trace Compass GUI. Here, I show a couple of hardware interrupts (IRQ lines 1 and 130, the interrupt lines for the i8042 and Wi-Fi chipset, respectively, on my native x86_64 Linux system):



T: 2020-07-30 09:41:31.298866556 T1: 2020-07-30 ...7-30 09:41:12.015277178 Δ: 0.000003238s (3.2 µs)

Figure 14.7: Sample screenshot of the Trace Compass GUI; samples recorded by LTTng showing IRQ lines 1 and 130

The pink color (we provide a PDF file that has full-color images of all the screenshots/diagrams used in this book. Link: `http://www.packtpub.com/sites/default/files/downloads/9781803232225_ColorImages.pdf`) in the upper part of the preceding screenshot represents the occurrence of a hardware interrupt. Underneath that, in the IRQ vs Time tab (it's only partially visible), the interrupt distribution is seen. (In the distribution graph, the *y* axis is the time taken; interestingly, the network interrupt handler – in red – seems to take very little time. The i8042 keyboard/mouse controller chip's handler – in blue – takes more time, even – in some cases – exceeding 200 microseconds.)

Disclaimer: Among the next few utilities/projects mentioned, I am the primary author of the `procmap` and SEALS project efforts.

The procmap utility

Visualizing the complete memory map of the kernel **Virtual Address Space** (**VAS**) as well as any given process's user VAS is what the `procmap` utility is designed to do.

The description on its GitHub page sums it up:

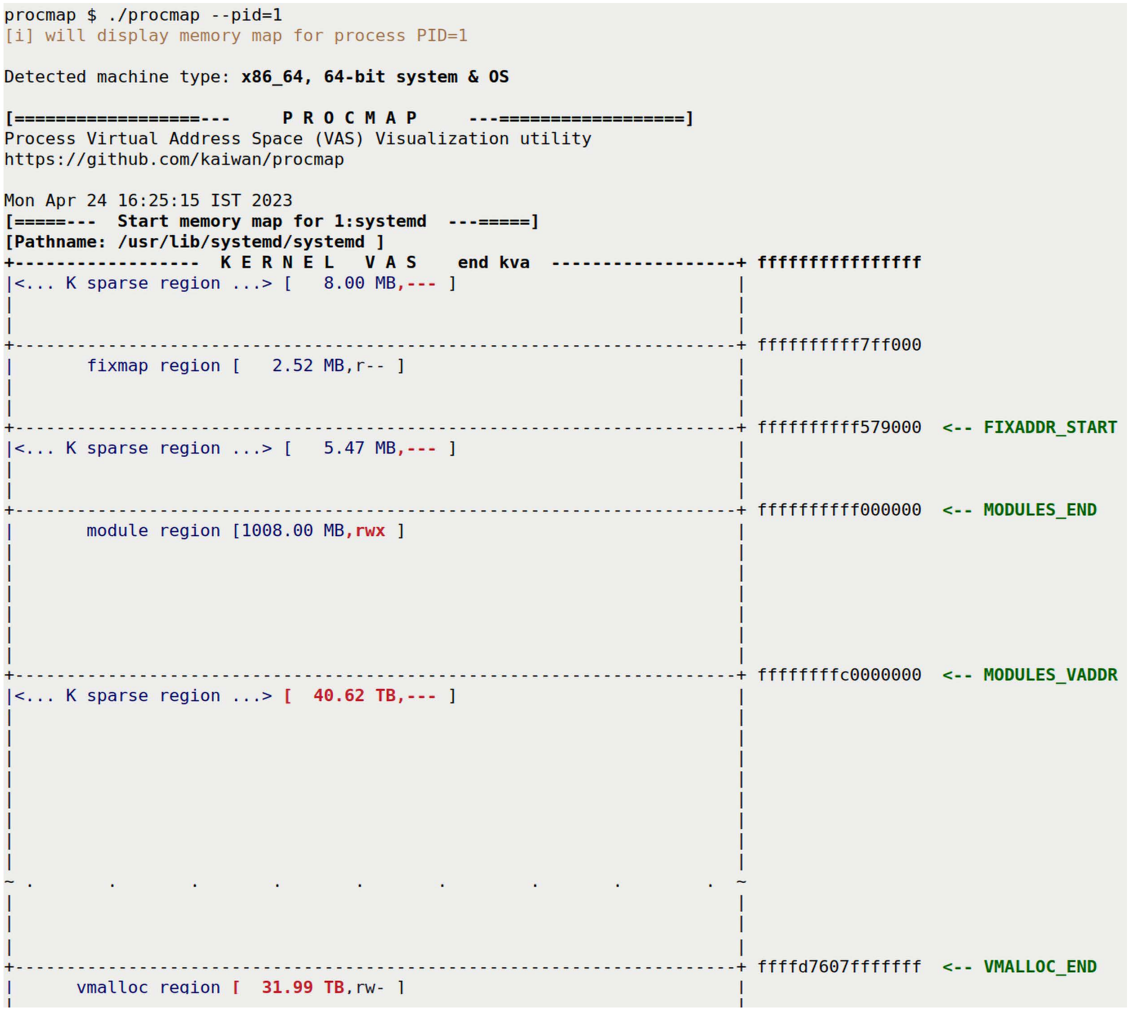
"It outputs a simple visualization of the complete memory map of a given process in a vertically-tiled format ordered by descending virtual address... The script has the intelligence to show kernel and user space mappings as well as calculate and show the sparse memory regions that will be present.

Also, each segment or mapping is (very approximately) scaled by relative size and color-coded for readability. On 64-bit systems, it also shows the so-called non-canonical sparse region or 'hole' (typically close to a whopping 16,384 PB on the x86_64)."

The utility includes options to see only kernel space or user space, verbose and debug modes, and the ability to export its output in convenient CSV format to a specified file. It has a kernel component as well and currently works (and auto-detects) on x86_64, AArch32, and AArch64 CPUs.

Do note, though, that this utility is still considered to be under development; there are several caveats (since the first edition, several fixes have been made; see the GitHub repo for details). Feedback and contributions are most appreciated!

Download/clone it from `https://github.com/kaiwan/procmap`. Here's (*Figure 14.8*) a partial screenshot of a run by procmap, showing a portion of its output (the VAS of the kernel):



```
procmap $ ./procmap --pid=1
[i] will display memory map for process PID=1

Detected machine type: x86_64, 64-bit system & OS

[===================---     P R O C M A P     ---===================]
Process Virtual Address Space (VAS) Visualization utility
https://github.com/kaiwan/procmap

Mon Apr 24 16:25:15 IST 2023
[=====---  Start memory map for 1:systemd  ---=====]
[Pathname: /usr/lib/systemd/systemd ]
+------------------  K E R N E L   V A S   end kva  ------------------+ ffffffffffffffff
|<... K sparse region ...> [   8.00 MB,--- ]                          |
|                                                                     |
|                                                                     |
+---------------------------------------------------------------------+ ffffffffff7ff000
|       fixmap region [   2.52 MB,r-- ]                                |
|                                                                     |
|                                                                     |
+---------------------------------------------------------------------+ ffffffffff579000  <-- FIXADDR_START
|<... K sparse region ...> [   5.47 MB,--- ]                          |
|                                                                     |
|                                                                     |
+---------------------------------------------------------------------+ ffffffffff000000  <-- MODULES_END
|       module region [1008.00 MB,rwx ]                                |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
+---------------------------------------------------------------------+ ffffffffc0000000  <-- MODULES_VADDR
|<... K sparse region ...> [  40.62 TB,--- ]                          |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
|                                                                     |
~ .       .       .       .       .       .        .       .        . ~
|                                                                     |
|                                                                     |
|                                                                     |
+---------------------------------------------------------------------+ ffffd7607fffffff  <-- VMALLOC_END
|      vmalloc region [  31.99 TB,rw- ]                                |
```

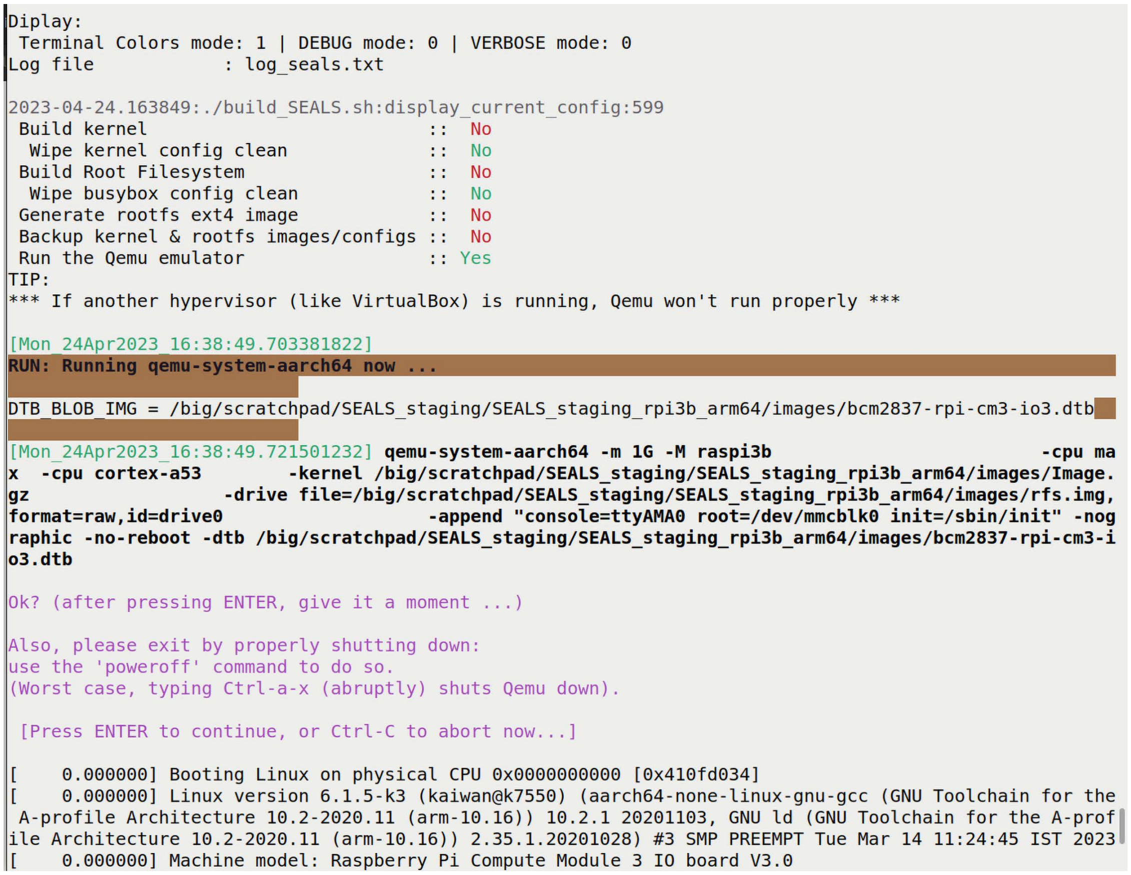
Figure 14.8: A partial screenshot of the procmap utility's output, showing only the top portion of kernel VAS on x86_64

We make good use of this utility in *Chapter 7*, *Memory Management Internals – Essentials*.

Simple Embedded ARM Linux System (the SEALS) project

SEALS, or **Simple Embedded ARM Linux System**, is a very simple “skeleton” Linux base system running on a QEMU-emulated (ARM) machine. It provides a primary Bash script that asks the end user what functionality they want via a menu, then accordingly proceeds to cross-compile a Linux kernel for ARM, and then creates and initializes a simple root filesystem. It can then call upon QEMU to emulate and run the chosen platform (the ARM-32 Versatile Express CA-9 is the default board emulated). The useful thing is, the script builds the target kernel, the root filesystem, and the root filesystem image file, and sets things up for boot. It even has a simple GUI (as well as a console-based) frontend, to make usage a bit simpler for the end user.

Since the first edition of this book, this project has undergone several fixes and enhancements; it can now emulate both 32- and 64-bit ARM platforms as well as the x86_64 PC platform. Here’s a partial screenshot of using SEALS to emulate a Raspberry Pi CM 3 IO board; you can just about see the (emulated) system booting up:



```
Diplay:
 Terminal Colors mode: 1 | DEBUG mode: 0 | VERBOSE mode: 0
Log file            : log_seals.txt

2023-04-24.163849:./build_SEALS.sh:display_current_config:599
 Build kernel                          ::  No
  Wipe kernel config clean             ::  No
 Build Root Filesystem                 ::  No
  Wipe busybox config clean            ::  No
 Generate rootfs ext4 image            ::  No
 Backup kernel & rootfs images/configs ::  No
 Run the Qemu emulator                 :: Yes
TIP:
*** If another hypervisor (like VirtualBox) is running, Qemu won't run properly ***

[Mon_24Apr2023_16:38:49.703381822]
RUN: Running qemu-system-aarch64 now ...

DTB_BLOB_IMG = /big/scratchpad/SEALS_staging/SEALS_staging_rpi3b_arm64/images/bcm2837-rpi-cm3-io3.dtb

[Mon_24Apr2023_16:38:49.721501232] qemu-system-aarch64 -m 1G -M raspi3b                         -cpu ma
x  -cpu cortex-a53        -kernel /big/scratchpad/SEALS_staging/SEALS_staging_rpi3b_arm64/images/Image.
gz                 -drive file=/big/scratchpad/SEALS_staging/SEALS_staging_rpi3b_arm64/images/rfs.img,
format=raw,id=drive0                   -append "console=ttyAMA0 root=/dev/mmcblk0 init=/sbin/init" -nog
raphic -no-reboot -dtb /big/scratchpad/SEALS_staging/SEALS_staging_rpi3b_arm64/images/bcm2837-rpi-cm3-i
o3.dtb

Ok? (after pressing ENTER, give it a moment ...)

Also, please exit by properly shutting down:
use the 'poweroff' command to do so.
(Worst case, typing Ctrl-a-x (abruptly) shuts Qemu down).

 [Press ENTER to continue, or Ctrl-C to abort now...]

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 6.1.5-k3 (kaiwan@k7550) (aarch64-none-linux-gnu-gcc (GNU Toolchain for the
 A-profile Architecture 10.2-2020.11 (arm-10.16)) 10.2.1 20201103, GNU ld (GNU Toolchain for the A-prof
ile Architecture 10.2-2020.11 (arm-10.16)) 2.35.1.20201028) #3 SMP PREEMPT Tue Mar 14 11:24:45 IST 2023
[    0.000000] Machine model: Raspberry Pi Compute Module 3 IO board V3.0
```

Figure 14.9: Partial screenshot showing the SEALS project run, emulating a Raspberry Pi CM3 via QEMU

Thus, SEALS can be useful to quickly try things out, for quick prototyping and learning. You don't even require actual hardware; your laptop is now sufficient (to emulate these boards and try things out)!

The project's GitHub page can be found here: `https://github.com/kaiwan/seals/`. Clone it and give it a try. A tutorial section helps you get off the ground quickly: `https://github.com/kaiwan/seals/#a-very-brief-tutorial-on-getting-going-with-seals`.

Modern tracing and performance analysis with eBPF

An extension of the well-known **Berkeley Packet Filter**, or **BPF**, is **eBPF**, the **extended BPF**. (FYI, at times it's referred to simply as BPF, dropping the "e" prefix; here we'll explicitly use the term eBPF.) Very briefly, BPF used to provide the supporting infrastructure within the kernel to effectively trace network packets. eBPF is a relatively recent kernel innovation – available only from the Linux 4.0 kernel onward. It extends the BPF notion, allowing you to trace much more than just the network stack. *eBPF is essentially virtual machine technology, allowing one to write (small) programs and run them in a safe, isolated environment within the kernel*. In effect, eBPF and its frontends are a really modern and powerful approach to observability, tracing, performance analysis, and more on Linux systems, *even in production*.

To use eBPF, you will need a system with the following:

- Linux kernel 4.0 or later
- Kernel support for BPF (`https://github.com/iovisor/bcc/blob/master/INSTALL.md#kernel-configuration`)
- The **BCC** or `bpftrace` frontends installed (link to install them on popular Linux distributions: `https://github.com/iovisor/bcc/blob/master/INSTALL.md#installing-bcc`)
- Root access on the target system

Using the eBPF kernel feature directly is very hard, so there are several easier frontends to use. Among them, BCC and `bpftrace` are regarded as very useful. Check out the following link to a picture that opens your eyes to just how many powerful BCC tools are available to help trace different Linux subsystems and hardware: `https://github.com/iovisor/bcc/blob/master/images/bcc_tracing_tools_2019.png`.

You can install the BCC tools for your regular host or native Linux distro by reading the installation instructions here: `https://github.com/iovisor/bcc/blob/master/INSTALL.md`.

Why not on our guest Linux VM? You can, when running a distro kernel (such as an Ubuntu- or Fedora-supplied kernel). The reason: the installation of the BCC toolset includes (and depends upon) the installation of the `linux-headers-$(uname -r)` package; this `linux-headers` package exists only for distro kernels (and not for our custom 6.1 kernel that we shall often be running on the guest). There is a workaround: building the kernel with the kernel config `CONFIG_IKHEADERS=m` (you'll learn how to do this in *Chapter 2, Building the 6.x Linux Kernel from Source – Part 1*). In fact, part of the error message from the eBPF tooling spells this out: "*Unable to find kernel headers. Try rebuilding kernel with CONFIG_IKHEADERS=m (module) or installing the kernel development package for your running kernel version.*"

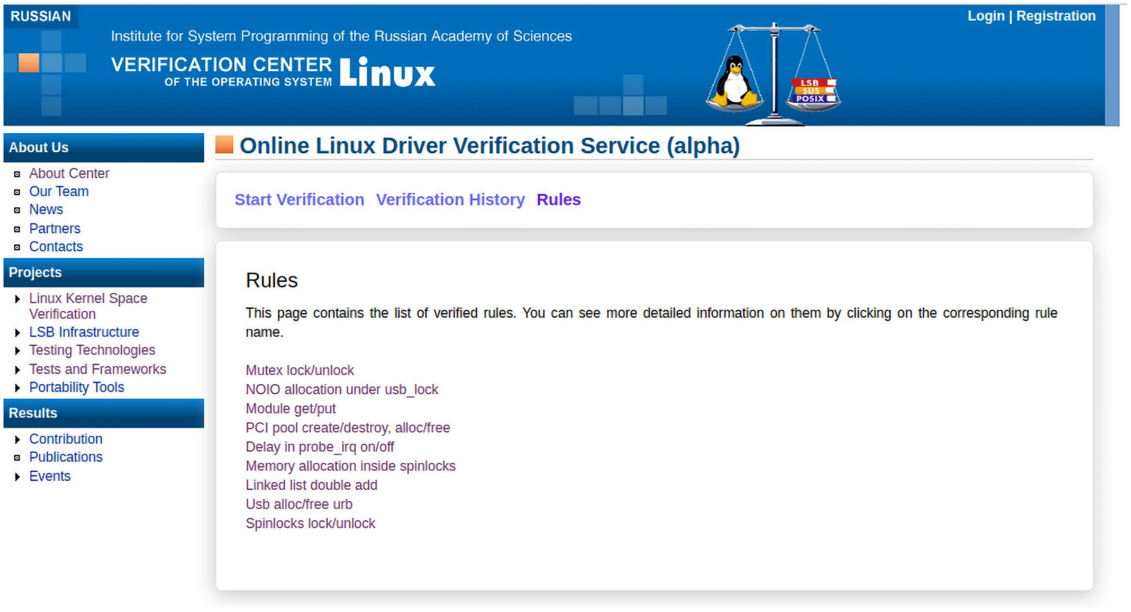
The main site for BCC can be found at `https://github.com/iovisor/bcc`. We shall make use of a bit of the eBPF tooling in some later chapters.

FYI, my book *Linux Kernel Debugging*, covers using LTTng, Trace Compass, KernelShark, ftrace, trace-cmd, procmap, and many more debugging tools and techniques in depth.

The LDV (Linux Driver Verification) project

The Russian Linux Verification Center, founded in 2005, is an open-source project; it has specialists in, and thus specializes in, automated testing of complex software projects. This includes comprehensive test suites, frameworks, and detailed analyses (both static and dynamic) being performed on the core Linux kernel as well as on the primarily device drivers within the kernel. This project puts a great deal of focus on the testing and verification of *kernel modules* as well, which many similar projects tend to skim.

Of particular interest to us here is the Online Linux Driver Verification Service page (`http://linuxtesting.org/ldv/online?action=rules`); it contains a list of a few verified rules (*Figure 14.10*):



Rules

This page contains the list of verified rules. You can see more detailed information on them by clicking on the corresponding rule name.

Mutex lock/unlock
NOIO allocation under usb_lock
Module get/put
PCI pool create/destroy, alloc/free
Delay in probe_irq on/off
Memory allocation inside spinlocks
Linked list double add
Usb alloc/free urb
Spinlocks lock/unlock

Figure 14.10: Screenshot of the Rules page of the Linux Driver Verification (LDV) project site

By glancing through these rules, we'll be able to not only see the rule but also instances of actual cases where these rules were violated by driver/kernel code within the mainline kernel, thus introducing bugs. The LDV project has successfully discovered and fixed (by sending in patches in the usual manner) several driver/kernel bugs.

In a few of the upcoming chapters, we shall mention instances of these LDV rule violations (for example, memory leakage, **Use After Free** (**UAF**) bugs, and locking violations) having been uncovered, and (probably) even fixed.

Here are some useful links on the LDV website:

- The Linux Verification Center home page: `http://linuxtesting.org/`
- Linux Kernel Space Verification: `http://linuxtesting.org/kernel`
- Online Linux Driver Verification Service page **with verified rules**: `http://linuxtesting.org/ldv/online?action=rules`
- *Problems in Linux Kernel* page; lists over 400 issues found in existing drivers (mostly fixed as well): `http://linuxtesting.org/results/ldv`

Without claiming to be complete by any stretch, perhaps more modern is tooling like the following:

- GitHub provides (optional, free to a limited extent for public repos) code-scanning workflow technology via GitHub Actions; see more details here: `https://github.com/features/actions`.
- Commercial: There are many suites for code-scanning; as one commercial example, Synopsys's Black Duck "software composition analysis (SCA) helps teams manage the security, quality, and license compliance risks that come from the use of open-source and third-party code in applications and containers."

Summary

In this chapter, we covered in detail the hardware and software requirements to set up an appropriate development environment for beginning to work on Linux kernel programming, including setting up a VM (via OSBoxes or otherwise). In addition, we mentioned the basics and provided links, wherever appropriate, for setting up a Raspberry Pi device, installing powerful tools such as QEMU (and a cross-toolchain), and so on. We also threw some light on other miscellaneous tools and projects that you, as a budding kernel and/or device driver developer, might find useful, as well as information on how to begin looking up the Linux `man` pages and Linux kernel documentation.

In this book, we recommend and expect you to try out and work on kernel code in a hands-on fashion – remember, always be empirical! To do so, you must have a proper kernel workspace environment set up, which we have successfully done in this chapter.

Now that our work environment is ready, let's move on and explore the brave world of Linux kernel development; your kernel journey's about to begin! The next two chapters will teach you how to download, extract, configure, and build a Linux kernel from source.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material: `https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/questions/ch1_qs_assignments.txt`. You will find some of the questions answered in the book's GitHub repo: `https://github.com/PacktPublishing/Linux-Kernel-Programming/tree/master/solutions_to_assgn`. The Q&A is organized chapter-wise.

Further reading

To help you delve deeper into the subject with useful materials, we provide a rather detailed list of online references and links (and at times even books) in a *Further reading* document in this book's GitHub repository. The *Further reading* document (again organized chapter-wise) is available here: `https://github.com/PacktPublishing/Linux-Kernel-Programming/blob/master/Further_Reading.md`.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/SecNet`