

The Linux Kaleidoscope

A Companion Handbook for Linux Kernel Programming

SUKRIT | PURU | ADITYA

The Linux Kaleidoscope

A Companion Handbook for Linux Kernel Programming

Version 0.1

November 2018

Sukrit, Puru, Aditya
{sukrit, puru}@cse.iitb.ac.in



Department of Computer Science of Engineering
Indian Institute Technology Bombay

Contents

1	Introduction	1
	Some Basics	1
	Some important resources	2
	Suggested Readings	2
	Online Resources	2
	Offline Resources	2
	Some things about the kernel	2
	Supplementary material	5
	Organization of this document	5
	Misc fun facts about the kernel	5
2	Getting Started	6
	Environment Setup	6
	Setup I: VirtualBox	7
	Setup II: virt-manager	7
	Setup III: qemu-system	7
	Setup IV: Bare Linux system	8
	Inside the VM	8
	Warming Up	9
	make	9
	Compiling the kernel from source	9
	Hello World!	12
	Patience!!	14
	Advanced	15
	make Jobs & CPU cores	15
	grep Is Your Friend	15
	rm-ing a Kernel	15
	config Pruning	16
	qemu Debugging	16
	ccache Optimization	16
	tags For Navigation	17
	Kernel Coding Style	17
3	System Calls	18
4	Modules	23
	Compiling a Module	24
	Loading/Unloading	25
	Passing command-line arguments	26
	Auto-loading Modules	27
	Exercises	29
	General	29
	Modules	29

5	Design Patterns	32
	Data Types	32
	Data Structures	32
	Lists	32
	Circular doubly linked list	32
	Hash lists	33
	Red-black Trees	33
	Radix Trees	34
	Kernel Log Buffer	34
	printk	35
6	Processes	37
	Suggested readings	37
	Books	37
	Others	37
	Introduction	37
	Structs and Flags	38
	List of important functions and macros	45
	Exercises	45
7	Memory	49
	Address Space Layout Randomization (ASLR)	51
	RAM Organisation	52
	Memory Allocators	53
	Page Allocator	53
	Page Allocator APIs	54
	Slab Allocator	56
	Slab Allocator APIs	57
	Structs and Flags	58
	Functions and Macros	60
	Assignments	60
	Suggested Reading	61
	menuconfig / Kconfig	62
	FLAGS	62
	processes	62

Preamble

```
$ ls -l linux-6.1.8.tar.xz
-rw-rw-r-- 1 abc xyz 129M Jan 26 10:22 linux-6.1.8.tar.xz

$ cloc linux-6.1.8.tar.xz
 78668 text files.
 78117 unique files.
 11213 files ignored.

github.com/AlDanial/cloc v 1.82 T=75.35 s (895.6 files/s, 445703.3 lines/s)
```

Language	files	blank	comment	code
C	32036	3247854	2563386	16703497
C/C++ Header	23313	699814	1332985	6903609
reStructuredText	3243	157809	62821	431622
JSON	485	2	0	389030
YAML	2981	54274	13666	247930
Assembly	1318	47180	100562	228246
Bourne Shell	884	26811	18621	104753
make	2780	10816	11822	49545
SVG	74	90	1171	48177
Perl	66	7381	5028	36525
Python	154	6527	6088	32631
yacc	9	697	409	4904
Rust	29	817	4687	4855
PO File	5	791	918	3077
lex	9	346	309	2108
C++	10	372	138	2016
Bourne Again Shell	54	392	321	1599
awk	13	217	157	1323
Glade	1	58	0	603
NAnt script	2	155	0	540
Cucumber	1	34	58	196
TeX	1	6	74	156
Windows Module Definition	2	15	0	109
m4	1	15	1	95
CSS	2	35	37	90
XSLT	5	13	26	61
MATLAB	1	17	37	35
vim script	1	3	12	27
Ruby	1	4	0	25
Markdown	1	8	0	25
INI	1	1	0	6
sed	1	2	5	5
TOML	1	1	9	2
SUM:	67485	4262557	4123348	25197422

This is what Linux looks like—a 129 MB source tar file and over tens of millions of lines of code.

First, let's clarify what this book is not,

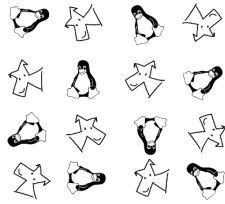
- it is not a textbook. definitely, not the first book you should be reading about operating systems.

- it is not a definitive guide about all things Linux.

-

The book is an attempt to

... Something about encouraging the students to read the source code....



1

Introduction

Some Basics

What is Linux?

You have probably heard this word way many times by now: Linux, and must be wondering what exactly is this?

Well, to be accurate, Linux is a family of operating systems that use the Linux Kernel at its core. So, that means that Linux is not just one OS, and many OSes can be referred to as Linux!

In fact, these operating systems, that are part of this family are called Linux Distributions ("distros" for short). Some of the most popular ones (purely opinion based, no hard feelings!) are Ubuntu, Fedora, and Arch Linux.

You see? Linux is like coffee, it has many flavours!

What is a Kernel?

But, then what is Linux Kernel? From OS 101, we all know that the heart of any operating system lies at its kernel. What you get to see when your OS boots up are just layers built upon the kernel.

As you would have probably guessed by now, Linux Kernel is what runs a Linux Distro.

Why is it called Linux? Well, long story short, Linux was "unintentionally" named after its main contributor, Linus Torvalds.

From now on, the word kernel in this handbook would imply Linux Kernel.

What does Open Source / FOSS mean?

Open source simply means that the source code of a project is publicly available for everyone to view/-modify/copy/etc.. You can see all the C/C++/Java/etc program files that make up that project and can see how things are getting done.

Nevertheless, certain implications imposed by licenses restrict your freedom in commercial usage and distribution of the code.

You must be wondering why make your project open for everyone else to see, right? I mean, it is your own cool idea upon which you are developing something and you would want it to be private. But, just imagine if Linus Torvalds had thought the same in 1991 and never released project he was working on.

If you ever want to be really good at programming in a language, read a few source files of a popular open-source project written in that language.

Oh! Did I tell you that the kernel is also open-source?

Some important resources

Suggested Readings

We will be referring to these awesome books on the Linux kernel throughout this handbook. It will greatly help if you read the relevant sections from these beforehand.

- *Linux Kernel Development* [lkd] by Robert Love
- *Understanding the Linux Kernel* [ulk] by Daniel P. Bovet and Marco Cesati
- *Linux Device Drivers* [ldd] by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
- *Professional Linux kernel architecture* [plka] by Wolfgang Mauerer

Online Resources

- The Linux Kernel Documentation [link] This is the official Linux Kernel documentation and is maintained by the kernel development community.
- The Linux Kernel Archives [link] The place where the Linux kernel source officially resides.
- Elixir Cross Referencer [link] If you want to browse the kernel code from the comfort of your web browser, Elixir is the place to go.
- Kernel Newbies [link] This is the place for Linux kernel learners and potential contributors.
- LWN (fka Linux Weekly News) [link] You can find the latest kernel development news here. It also features great articles related to kernel hacking.

Offline Resources

Now that you have the kernel source with you, you have the most important resource at hand! The kernel provides some documentation in the form of plain text files which is located inside the directory `Documentation`.

The same is also available online [here].

There will also be times when you need to find details about a Linux command or a glibc function. You can always use Linux man pages using the `man` command.

Some things about the kernel

The kernel is coded primarily in C (C90 to be precise) and Git is used for its version control.

Git? Sounds familiar?

It is another software marvel from the creator of Linux himself! Also powered by the open source community, Linus made Git in 2005 to manage Linux kernel development!

While we won't be covering anything related to Git in this handbook, you are encouraged to learn Git if you are looking forward to doing Linux system programming long-term.

Where to find the Kernel source?

You can find the Linux kernel at <https://www.kernel.org>. If you need a specific version, you can get it from <https://mirrors.edge.kernel.org/pub/linux/kernel/>.

If you know Git and the benefits that come with it, you can clone the stable repository:

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
cd linux
git checkout tags/v4.17.4
(optional) git checkout tags/v4.17.4 -b mybranch
```

NOTE: Cloning this repo may take quite a while as this method will download around 2 GB of data, which is a lot more compared to the tar files.

You will notice there are two formats in which the compressed kernel is available: tar.gz and tar.xz. xz is based on the LZMA2 compression algorithm. It achieves a compression ratio higher than that of gz format but takes more time to [de]compress. The xz compressed kernel source tarball is 97 MB in size. gz is based on DEFLATE compression algorithm. It achieves a lower compression ratio, but is quicker to [de]compress. The gz compressed kernel source tarball is 150 MB in size. git vs. tar.xz vs. tar.gz, choice is yours, totally!

Kernel version numbering

For the kernel before 1.0, incremental numbers denoted a release. For example, the first kernel version was 0.01 (released in September 1991), then the next kernel version was 0.02, and so on.

After 1.0 and up to 2.x, the kernel versions had a number pattern with deeper meaning. If the kernel version was a.b.c.d, then:

- a specified the (major) version
- b specified the minor version / patch level / major revision (odd for development, even for stable)
- c specified the (minor) revision number or sublevel
- d specified patch version for bug fixes and security patches

Kernel 2.6.x series is the longest running series till date. It went on from 2.6.0 to 2.6.39.

Since the kernel 3.x series, there are only three numbers in play, a.b.c where a was the version, b was the revision and c denoted the patch version. The kernel version numbers would be followed by some characters such as -rc1, which resembled that the kernel is a "release candidate", i.e. development version.

Kernel release types

- Release Candidate (RC)
- Mainline
- Stable
- Longterm

If you are ever confused about which kernel version to use, read [What stable kernel should I use?](#).

We'll use kernel 4.17.4, because it is latest (was, at the time of creation of this handbook!) and has some really cool features.

Kernel 4.17 series is code-named "Merciless Moray" (moray is a predatory eel-like fish!). Mentioned below

are few of the many new features and changes introduced in the kernel version 4.17, not that we will make use of any of these. For a complete list of changes (with links to respective commits), visit [here](#). Also visit [here](#) for a brief summary of kernel version changelogs.

- Improving the idle power consumption (by 10%) by reworking the kernel's idle loop.
- Removed support for some unused architectures such as blackfin, cris, frv, m32r, metag, mn10300, score and tile while adding support for nds32.
- Various improvements in the DRM drivers of Intel and AMD.
- Enhanced protection against Meltdown and Spectre.
- TLS socket implementation, which allows usage of TLS sockets from within the kernel.

NOTE: The kernel which we are using for our purpose is a "vanilla" kernel, i.e. it is fetched from the official kernel development tree without any modifications. There are some Linux distributions which ship a modified version of kernel (called vendor-supplied kernel), containing some patches relevant to that distribution.

Kernel source code structure

<https://elixir.bootlin.com/linux/v4.17.4/source>

Subsystems

- arch
- block
- crypto
- drivers
- firmware
- fs
- include
- init
- ipc
- kernel
- lib
- mm
- net
- samples
- scripts
- security
- sound
- usr
- tools
- virt

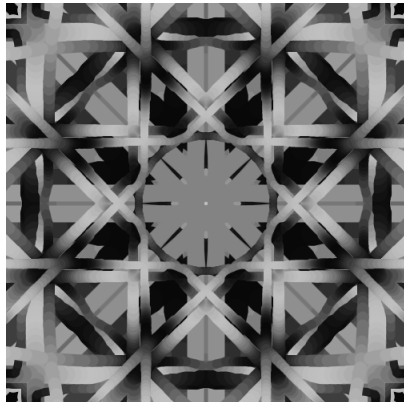
Supplementary material

Our GitHub repository [kaleidoscope](#) contains the source code for the exercises provided with each chapter.

Organization of this document

Misc fun facts about the kernel

- More than 50% of the code in the Linux Kernel is in drivers/.
- The longest file in the kernel belongs to AMD DRM driver (`drivers/gpu/drm/amd/include/asic_reg/nbio/nbio_6_1_sh_mask.h`), having 111547 lines of code and is 14MB in size! Good luck opening it in a text editor!
- All the supercomputers in the TOP500 list, as of November 2017, ran on Linux. IBM's Summit and Sierra, the faster supercomputers in the world as of November 2018 also run on Linux.



2

Getting Started

Environment Setup

So, you want to start kernel development? Worry do not. Learn you will.

You would need to set up a few things before you can go on. These include, but are not limited to, base machine, OS, development tools and some configurations.

Throughout this journey, we will be using the following tools:

- Ubuntu Server 20.04.05 LTS
- Vim
- GCC

If you understand the concept of a virtual machine and know why it is useful in our scenario, you can skip to the next subsection.

<Proper reasoning for why VM?> Now, we are talking about hacking on the kernel, and that would require modifying some of its source files and maybe adding some new files. A kernel forms the core of your OS. Playing with your (live) OS is not really a good idea, unless you have bought a dedicated machine for this purpose. You might have a browser open where you are browsing stuff, a media player open where you are listening to some music while coding, etc. Kernel development would require sporadic reboots of your system. To break free from these infrastructural problems, we will harness the power of modern day virtualization and do all our hacking inside a Virtual Machine (VM).

Please note that your processor must support virtualization in order for our plan to work out. If not, you have to use your bare machine for development.

To check virtualization support on a Linux system: Open a terminal and enter:

```
egrep "vmx|svm" /proc/cpuinfo
```

To check virtualization support on a Windows system: Open the command prompt and enter:

```
systeminfo
```

To enable virtualization if it is supported by your processor, check BIOS settings.

NOTE: Once you install your OS in a VM, and finish all the basic environment setup tasks, do not use it. It can be your base VM. If you need a VM to work on, simply copy/clone the base VM. This will save you a LOT of time and resources.

Setup I: VirtualBox

VirtualBox is a hosted virtualization solution by Oracle. It is FOSS and written primarily in C++. Apart from that, it is easy to set up and is available for Windows, macOS and most of the Linux distributions.

This is recommended for Windows and macOS users. They do not have any other easy option!

You can download the software from the official website: <https://www.virtualbox.org>

On ubuntu:

```
sudo apt install virtualbox virtualbox-ext-pack
```

May not be the latest release though!

Setup II: virt-manager

This is for Linux users who want to dive in real quick.

```
sudo apt install virt-manager qemu-kvm
```

It will install libvirt-bin as well.

Setup III: qemu-system

This one is for those who want to pursue kernel programming for eternity, more or less.

Host steps:

> Install required package

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

> Create an image for VM

```
qemu-img create -f qcow2 <path to VM img> 30G
```

This will create an image (virtual hard disk for the VM) of size 30 GB.

> Install Ubuntu in VM

```
sudo qemu-system-x86_64 \
    -enable-kvm \
    -m 1024 \
    -smp cpus=2 \
    -hda <path to VM img> \
    -cdrom <path to iso>
```

This will start an "empty" VM with 1 GB RAM, 2 CPU cores, using KVM support for virtualization and the disk you created in the previous step. You only need to provide the path to the .iso for the very first time.

> Run the VM

```
sudo qemu-system-x86_64 \
    --enable-kvm \
    -m 1024 \
    -smp cpus=2 \
    -hda <path to VM img> \
    -net nic \
    -net user,hostfwd=tcp::2222-:22
```

The last line sets up port forwarding so that all communication on the hosts port 2222 gets sent to the VMs port 22. `-net nic` initialises a virtual network interface card. This lets you use SSH to login from your host system.

```
ssh localhost -p 2222
```

This way, you can use your favourite terminal and basic things like copy-paste will be easier.

NOTE: If you plan on experimenting with different versions of kernel, make sure that you have a separate partition for /boot and give at least 1 GB space to it. The kernel that we will compile initially will take around 400 MB space. Plan accordingly.

Setup IV: Bare Linux system

The only benefit about this is that you don't need to do any setup for the host machine as the previous approaches did.

Inside the VM

By now, you must have a VM ready with an OS installed. Everything that we do from this point on will work regardless of what VM you use, be it VirtualBox, virt-manager or QEMU, as long as you are running Ubuntu.

Initial setup of the system:

```
sudo apt update
sudo apt upgrade
<reboot>
```

Install packages required by the compilation process

```
sudo apt install gcc make bash binutils flex bison pahole util-linux kmod build-essential
libncurses5-dev libelf-dev libssl-dev bc flex bison
```

Download the kernel source

```
https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.8.tar.xz
```

Extract the files

```
tar -xvf linux-6.1.8.tar.xz
cd linux-6.1.8/
```

Warming Up

Before we dive into the internals of the kernel, let us get our hands on some code.

make

GNU make (or just "make") is a development tool used by the kernel (and by most of the open-source projects) to compile all the source files and produce executables.

A file named `Makefile` resides in the root directory of a project. It contains a set of directives which specify the structure in which the source files are to be compiled. The make utility reads this file by default to proceed with the compilation.

In the following sections, you will frequently run commands of the form `make xyz`, where `xyz` is a make target.

Compiling the kernel from source

You should be ready with your environment setup to proceed with this. A high-level overview of the steps to be followed to compile and build a custom kernel is given below.

1. Create a configuration file. The configuration file, `.config` is stored in the root of the kernel source tree and used by the kbuild system to compile and build the kernel.
2. Compile and build the kernel modules using `make -j$(nproc)` which will be stored in `arch/x86/boot/`. This generates the compressed kernel image `bzImage`, the uncompressed kernel image `vmlinux`, the Symbol-address table `System.map`, and kernel module objects. `vmlinux` and `System.map` are useful for debugging purposes (during boot, the GRUB bootloader uncompresses the compressed kernel image and jumps to the kernel's entry point).
3. Install the newly built kernel modules to a default location (which is typically `/lib/modules/$(uname -r)`) using `sudo make modules_install`.
4. Create the `initramfs` image and update the `GRUB` menu using `sudo make install`.
5. Reboot into your newly built kernel!

Go to the directory of kernel source

```
cd linux-6.1.8
```

Create configuration file The entire configuration and building of the Linux kernel is managed by the kernel's Kbuild system. The Kbuild system consists of four components:

- Configuration symbols: Any option that can be modified in the kernel (like the size of the kernel log buffer) is represented by a configuration symbol.
- Kconfig files: Configuration symbols are defined in Kconfig files. It turns out that the contents of Kconfig files are used to generate menu entries when `make menuconfig` is used to modify the configuration.
- Makefile: The Kbuild system uses a recursive Makefile structure to build the kernel source. During the build process, the top-level Makefile invoking Makefiles in sub-folders.

- The `.config` file: On generating the configuration, all the symbols are stored in the `.config` which is located in the root of the source tree.

The kernel has just way too many configuration options! It's hard to think about how one could assign values to each and every one of these. Fortunately, there is a workaround. We can start by creating a default configuration and modifying that to suit our requirements. There are a couple of ways to do this, we just list some of the more commonly used ones.

Distribution Config

You can use your distribution's `config` file which can be found in `/lib/modules/$(uname -r)/`. Copy it to the root of the source tree and run `make menuconfig` to modify the configuration:

```
cd linux-6.1.8
cp /lib/modules/$(uname -r)/.config .
make menuconfig
```

Using currently loaded modules

Another way of obtaining a good configuration is to base it off the kernel modules current running on the system. You can list the modules loaded using `lsmod` and redirect the output to a file. This file is passed to `localmodconfig`:

```
lsmod > current_modules
make LSMOD=current_modules localmodconfig
```

Since the kernel source is updated frequently, it is very likely that there are differences in configuration options between the kernel you are trying to build and the one you are currently running. In such cases, the `kbuild` system will prompt you to select values for new configuration symbols. To keep things simple, just press `<ENTER>` and proceed (this assigns default values to the configuration symbols). For further customisation, you can run `make menuconfig` to finetune the kernel configuration. As an example, disable Device Drivers -> Graphics Support -> Direct Rendering Manager in the `menuconfig` UI.

NOTE: You can also disable configurations using the `config` script as follows,

```
cd linux-6.1.8
scripts/config --disable CONFIG_DRM
```

Compile and build the kernel and modules.

```
make -j$(nproc) > /dev/null 2>&1
OR
screen -dm make -j$(nproc)
OR
tmux
make -j$(nproc)
```

NOTE: In case you don't have `screen` or `tmux`, you can install them using,

```
sudo apt install screen tmux
```

If the command executes successfully, you should be able to see the `vmlinux` and `System.map` file in the root of the source tree. You should also be able to see the compressed kernel image `bzImage` in the folder `arch/x86/boot`. Finally, the source tree should have been populated with compiled kernel object (`.ko`) files. To find the modules generated, you can run the following command:

```
find . -name "*.ko"
find . -name "*.ko" | wc -l
```


The second command counts the number of kernel modules.

NOTE: An error you might face during the build process is the following:

```
make[1]: *** No rule to make target 'debian/canonical-certs.pem', needed by 'certs/
x509_certificate_list'. Stop.
make: *** [Makefile:1809: certs] Error 2
```

This error has to do with the fact that kernel versions post 5.14-rc2 require all kernel modules to be signed for security purposes. The kernel module signing facilities signs all modules during installation and validates the signature during loading. If there is a signature mismatch, the module is not loaded into the kernel. The `CONFIG_MODULE_SIG_KEY` configuration contains the name of a module signing key. If it is unchanged from the default value of `certs/signing_key.pem`, a public-private keypair is automatically generated during the build process (the public key is built into the kernel during the build process). `CONFIG_SYSTEM_TRUSTED_KEYS` gives you the ability to include additional certificates that the kernel will use for signature checking. `CONFIG_SYSTEM_REVOCATION_KEYS` is an option that allows you to blacklist certain keys, i.e, any module signed with these keys will not be loaded into the kernel.

By default, the configuration `CONFIG_SYSTEM_REVOCATION_KEYS` will be set to `debian/canonical-certs.pem`. If this file is not present in the kernel source tree, `make` will throw the above error. Similarly, the symbol `CONFIG_SYSTEM_TRUSTED_KEYS` might be set or unset. To resolve the error, you can either disable both configurations:

```
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS
```

Or, you can download the required `.pem` file using,

```
sudo apt install linux-source
mkdir debian
cp /usr/src/linux-source-*/debian/canonical-*.pem debian/
sudo apt purge linux-source*
```

and set the symbols to,

```
scripts/config --set-str SYSTEM_TRUSTED_KEYS debian/canonical-certs.pem
scripts/config --set-str SYSTEM_REVOCATION_KEYS debian/canonical-revoked-certs.pem
```

Install the kernel modules in `/lib/modules/$(uname -r)`

```
sudo make modules_install -j$(nproc) > /dev/null
```

Simply building the kernel modules is not enough! The modules need to be installed to a known location in the root filesystem so that the system can locate them and load them into kernel memory. The default location for this is `/lib/modules/$(uname -r)`. However, you can override this preference by installing the modules to a location you specify:

```
make INSTALL_MOD_PATH=<path here> modules_install -j$(nproc)
```

Generate the `initramfs` image

```
sudo make install -j4 > /dev/null
```

Internally, the `make install` runs the script `arch/x86/boot/install.sh`. This script starts by creating an `initramfs` image which is used by the GRUB bootloader as a temporary file system during the boot process, following which the actual root filesystem is mounted using a pivot-root. After generating the image, it copies the files, `System.map-$(uname-r)`, `initrd.img-$(uname-r)`, `vmlinuz-$(uname-r)` and

`config-$(uname-r)` to the `/boot` folder in your system. Finally, the file `/boot/grub/grub.cfg` is updated with an entry for the newly-built kernel.

Update the GRUB entries to reflect the new kernel.

```
sudo update-grub2
```

Unfortunately, if you try to boot the system right now, you will not be given the option to boot into a kernel of your choice (by default, the bootloader will boot into the newest kernel). In fact, no menu will even be shown! This can be changed by making changes to the GRUB bootloader config file.

```
sudo cp /etc/default/grub /etc/default/grub.old
sudo vim /etc/default/grub
```

The first command makes a copy of the original configuration and the second opens the new configuration in Vim. To show the GRUB menu at boot, change the `GRUB_TIMEOUT_STYLE` from `hidden` to `menu`. Run `sudo update-grub2` to bring the changes into effect.

Reboot and voila!

```
<reboot>
uname -r
```

If all the steps were done correctly, this should display the version of the kernel just compiled.

Hello World!

You do not need to compile a new kernel to start coding kernel modules. Following the *K&R Hello, World* tradition, here's a module that does just that:

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

MODULE_LICENSE("Dual MIT/GPL");
MODULE_AUTHOR("Sukrit Bhatnagar, Purushottam Kulkarni & Aditya Sriram");
MODULE_DESCRIPTION("Customary Hello, World module!");

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Type in or copy the above program into a file name `hello.c` and save it. In the same directory, create a file named `Makefile` with the following contents:

```
obj-m += hello.o
```

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

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

The run `make`, and you should see a file named `hello.ko` created in the same directory. Then run the following command to load the kernel module:

```
sudo insmod hello.ko
```

If the above command gets executed without any message printed, your module was successfully loaded into the kernel! Congratulations!

NOTE: For newer kernel versions, you might encounter the following error,

```
insmod: ERROR: could not insert module hello.ko: Key was rejected by service
```

Printing the contents of the kernel log buffer using `dmesg | less`, you should see the following message,

```
Loading of unsigned module is rejected
```

Again, this is due to the kernel signing facility. Since this is a user-built module, it is not signed during the installation process and must be signed manually. You can do this using the `sign-file` script in the `scripts` folder. Assuming the newly built module is a folder named `hello`, run the following command:

```
cd linux-6.1.8
scripts/sign-file sha512 certs/signing_key.pem certs/signing_key.x509 ../hello/hello.ko
```

The `sign-file` script requires the type of hash to be used along with a signing key. By default, `SHA512` is used as the hash for module signing and if you do not modify the `SYSTEM_TRUSTED_KEYS` configuration, the kernel build will automatically generate the signing key, which has been used above. Once you sign your module, running `modinfo` should display a module signature,

```
signer:      Build time autogenerated kernel key
sig_key:     04:1A:C5:9E:60:91:C3:73:F0:2C:BE:9D:D9:18:2C:BD:94:F6:7A:6A
sig_hashalgo: sha512
signature:   B5:10:CC:5A:EC:76:49:82:B0:E4:3D:0F:F6:F5:73:F8:54:0F:EB:72:
             85:BB:F9:79:AF:F8:FA:DB:E3:34:24:A1:7E:F4:72:7A:78:1F:11:E9:
             82:E7:E7:51:52:D0:24:F1:7B:B9:1A:BE...
```

If you are wondering why nothing got printed even when the module was loaded successfully, the answer is: something did get printed, but not on the terminal you are working on.

To find your module output, run the following command:

```
dmesg | less
```

And voila! Here is your "Hello, world!"

To unload the module you just loaded and print "Goodbye, world!", run:

```
sudo rmmod hello
```

Note that when removing the module, we did not include the `.ko` extension in the name. Now, run again:

```
tail /var/log/syslog
```

And you will see the other message.

Patience!!

Kernel programming is unlike any other type of C/C++ or system programming that you might have done. Here you are playing with the OS directly and therefore, some things would take time longer than usual. For instance, a typical first-time kernel compilation requires around 90 minutes! Also, if your module had a logical error which raised a kernel oops, you have to restart your machine or VM! All this requires a lot of patience to go on.

Advanced

Here are a few tools and techniques that will help you in speeding up things, but are not mandatory per se. You may skip this as per your convenience.

make Jobs & CPU cores

CPU cores matter when you compile the kernel from source. Having a few extra cores might help speed up kernel compilation. This is made possible using the `-j` or `-jobs` option of `make`. This option will let the user decide how many make jobs can run simultaneously, which would normally mean the number of cores to use. Having a good amount of make jobs will fasten the compilation process, something you will wish for!

The general idea is to run as many jobs as you have CPU cores available. So, if you have x cores, then you would want the `jobs` somewhere between x and $2 \times x$. You can find out the number of cores (available to you) easily using `nproc` command.

But we don't want easy now, do we? A more detailed description about your processing units will be available in `/proc/cpuinfo`.

grep Is Your Friend

Many a times it will happen that you need to search for a specific token in the kernel source, and you are too lazy to browse LXR. Moreover, it might happen that the kernel make target "tags" might not work always as expected.

`grep` will prove to be a great tool in such cases. For instance, if you need to look at the definition of `struct thread_info`, just run this in the kernel root:

```
grep -rI "^struct thread_info" .
```

If by any chance you have a multicore/multiprocessor system, and SMP is enabled in the kernel, you can use the following command for a significant speedup. You must have GNU parallel installed for this.

```
find . -maxdepth 4 -type f | parallel -k -j150% -n 1000 -m grep -I -H <string to search> {} | \
    grep --color <string to search>
```

rm-ing a Kernel

On the newer Ubuntu systems, you have the option of removing old and unused packages using our friend `apt`:

```
sudo apt autoremove
```

Using this method, you can remove unused kernel versions that are vendor-supplied (Ubuntu in this case).

To manually remove an installed kernel, first do a `uname -r` to ensure that the kernel you are deleting is indeed the right one. You wouldn't want to delete the wrong kernel now, would you?

A typical kernel make process copies kernel files at two locations: `/boot` and `/lib/modules`. Let's assume that you have installed kernel 4.17.4 and now wish to delete it because compiling a kernel has now become a hobby for you. You would want to delete the following:

- `/boot/config-4.17.4`
- `/boot/initrd.img-4.17.4`
- `/boot/System.map-4.17.4`
- `/boot/vmlinuz-4.17.4`
- `/lib/modules/4.17.4`
- `/var/lib/initramfs-tools/4.17.4`

TIP: In case you are deleting a compiled kernel to free up some space, do not forget to run a `make clean` in the kernel source directory.

config Pruning

When you are compiling the kernel, the first step is to make the `.config` file. By default the file has many modules and features enabled which you probably will never use, at least during the course of this handbook.

Furthermore, compiling a kernel with the default `.config` file leads to creation of a very large `initrd` image file, nearly 379 MB in size! A normal such image of the vendor-supplied kernel would take around 40 MB. See the difference.

As a result, your kernel compilation will take longer, and even your system will boot slower when using that compiled kernel.

It is generally a good idea to disable the configuration options not required at the moment.

qemu Debugging

ccache Optimization

Ccache is a development tool which caches object files created by a make execution. Upon recompilation, the cached files can be used which saves compilation time (a lot in case of kernel). The first compilation will take the original amount of time though!

```
sudo apt install ccache

#Append the following to your ~/.bashrc:
export PATH="/usr/lib/ccache:$PATH"
export CCACHE_DIR="/ramdisk/ccache" #default is ~/.ccache

ccache -M 2G

#Either logout/login or source ~/.bashrc
```

tags For Navigation

When you are browsing through the kernel source, you find that a function `x` calls function `y`. Now, you need to search for function `y` in the code to understand what happens next. One way to go is to visit Linux Cross Referencer and search. Another (sometimes more efficient) way is to make use of tags. Tags are an index of all the symbols (function, struct, macro, etc.) found in the source code.

Linux kernel source provides a make target to automatically generate tags to use with editors such as Vim.

To use these tags:

```
sudo apt install ctags
make tags -j8
```

Then, in Vim, move the cursor to a {function, struct, macro, etc.} and press `Ctrl+]`. Shazam! You are now standing at its definition!. To go back, press `Ctrl+t`. You can, similarly, chain the jumps one after another.

If you are using Vim, append the following line to your `$HOME/.vimrc`:

```
set tags=./tags,tags;$HOME
```

This options enables Vim to search for tags file, starting from the directory of the current file, then the parent directory, then recursively one level up till home directory of the current user is reached.

Furthermore, if you wish to search for a tag, simply open Vim in the kernel source directory and type the following command:

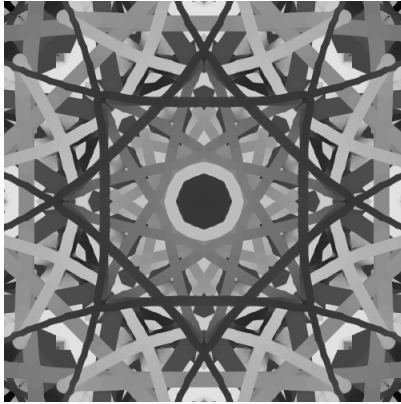
```
:tag <whatever>
```

Kernel Coding Style

... if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

This page contains some coding guidelines that are followed in the actual kernel development. Here is a very short summary of some of the basics:

- tab width: 8 characters; you code should not use more than 3 levels of nesting
- maximum line length: 80 characters; break lines if they exceed
- opening braces: at the end of line (except for function declarations)
- no mixed-case names; use underscores if required
- function names either denote an action (i.e. start with a verb) or they are a predicate
- use `-ve` values for failure, and `0` for success in functions doing some action
- use `'true'` for success and `'false'` for failure in predicate functions



3

System Calls

The operating system, just like any other system, stores some state and performs some actions. System calls are requests made by a program running in "user mode" to get something done by the kernel. They execute in kernel mode on behalf of a user process. Typically, the system calls you invoke from a C program are glibc wrapper of the real system call. In this chapter, we will not discuss what these system calls do. We will instead study how they do what they do.

The system calls are defined in `arch/x86/entry/syscall_64.c`. This file contains all the system calls that are supported by the system. The system calls headers are declared in `include/linux/syscalls.h`.

```
...
#define SYSCALL_DEFINE0(sname) \
    SYSCALL_METADATA(_##sname, 0); \
    asmlinkage long sys_##sname(void); \
    ALLOW_ERROR_INJECTION(sys_##sname, ERRNO); \
    asmlinkage long sys_##sname(void)
...
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)
...
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)
...
#define SYSCALL_DEFINEx(x, sname, ...) \
    SYSCALL_METADATA(sname, x, __VA_ARGS__) \
    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
...
#define __SYSCALL_DEFINEx(x, name, ...) \
    asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__)) \
    __attribute__((alias(__stringify(__se_sys##name)))); \
    ALLOW_ERROR_INJECTION(sys##name, ERRNO); \
    ...
```

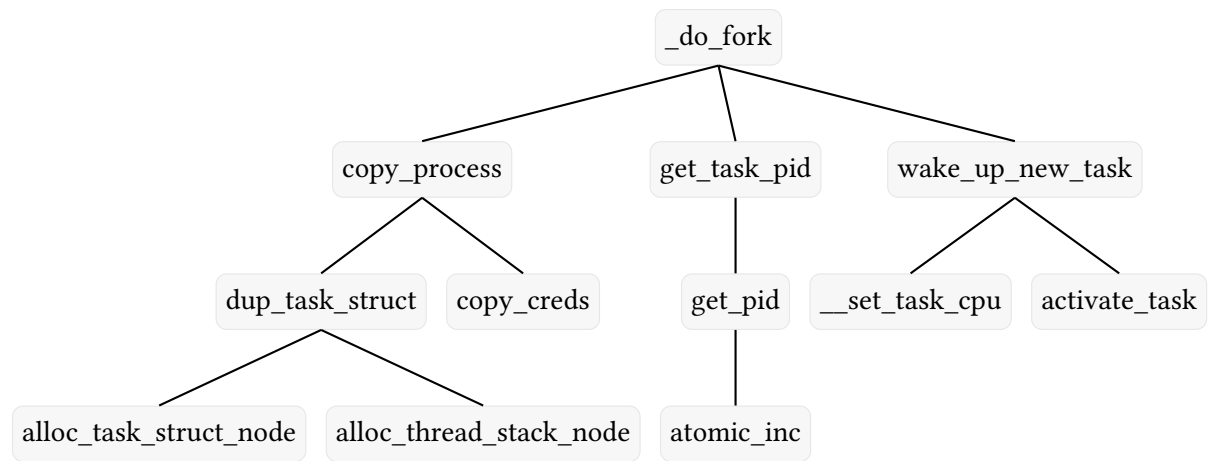
One of the good things about the kernel code is that almost all the internal functions are named starting with a verb. Example: `do_fork` which really does fork! A function's name is the essence of what it does.

Some important system calls related to process management: `fork`, `vfork`, `exec`, `wait`, `exit`

Some important system calls related to memory management: `shmget`, `sbrk`, `mmap`

The source for `fork` is in `kernel/fork.c`: `SYSCALL_DEFINE0(fork)`

`_do_fork()` is the function called!



We have included the complete list of error codes here, for the reader's reference. You can also find them in the `include/uapi/asm-generic` folder of the kernel source tree in the files, `errno-base.h` and `errno.h`.

EPERM	1	/* Operation not permitted */
ENOENT	2	/* No such file or directory */
ESRCH	3	/* No such process */
EINTR	4	/* Interrupted system call */
EIO	5	/* I/O error */
ENXIO	6	/* No such device or address */
E2BIG	7	/* Argument list too long */
ENOEXEC	8	/* Exec format error */
EBADF	9	/* Bad file number */
ECHILD	10	/* No child processes */
EAGAIN	11	/* Try again */
ENOMEM	12	/* Out of memory */
EACCES	13	/* Permission denied */
EFAULT	14	/* Bad address */
ENOTBLK	15	/* Block device required */
EBUSY	16	/* Device or resource busy */
EEXIST	17	/* File exists */
EXDEV	18	/* Cross-device link */
ENODEV	19	/* No such device */
ENOTDIR	20	/* Not a directory */
EISDIR	21	/* Is a directory */
EINVAL	22	/* Invalid argument */
ENFILE	23	/* File table overflow */
EMFILE	24	/* Too many open files */
ENOTTY	25	/* Not a typewriter */
ETXTBSY	26	/* Text file busy */
EFBIG	27	/* File too large */
ENOSPC	28	/* No space left on device */
ESPIPE	29	/* Illegal seek */
EROFS	30	/* Read-only file system */
EMLINK	31	/* Too many links */
EPIPE	32	/* Broken pipe */
EDOM	33	/* Math argument out of domain of func */
ERANGE	34	/* Math result not representable */
EDEADLK	35	/* Resource deadlock would occur */
ENAMETOOLONG	36	/* File name too long */
ENOLCK	37	/* No record locks available */
ENOSYS	38	/* Invalid system call number */
ENOTEMPTY	39	/* Directory not empty */
ELOOP	40	/* Too many symbolic links encountered */
EWouldBlock	EAGAIN	/* Operation would block */
ENOMSG	42	/* No message of desired type */
EIDRM	43	/* Identifier removed */
ECHRNG	44	/* Channel number out of range */
EL2NSYNC	45	/* Level 2 not synchronized */
EL3HLT	46	/* Level 3 halted */
EL3RST	47	/* Level 3 reset */
ELNRNG	48	/* Link number out of range */
EUNATCH	49	/* Protocol driver not attached */
ENOCSI	50	/* No CSI structure available */
EL2HLT	51	/* Level 2 halted */
EBADE	52	/* Invalid exchange */
EBADR	53	/* Invalid request descriptor */
EXFULL	54	/* Exchange full */
ENOANO	55	/* No anode */
EBADRQC	56	/* Invalid request code */

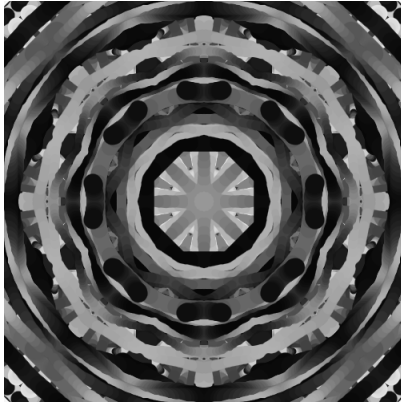
EBADSLT	57	/* Invalid slot */
EDEADLOCK	EDEADLK	
EBFONT	59	/* Bad font file format */
ENOSTR	60	/* Device not a stream */
ENODATA	61	/* No data available */
ETIME	62	/* Timer expired */
ENOSR	63	/* Out of streams resources */
ENONET	64	/* Machine is not on the network */
ENOPKG	65	/* Package not installed */
EREMOTE	66	/* Object is remote */
ENOLINK	67	/* Link has been severed */
EADV	68	/* Advertise error */
ESRMNT	69	/* Srmount error */
ECOMM	70	/* Communication error on send */
EPROTO	71	/* Protocol error */
EMULTIHOP	72	/* Multihop attempted */
EDOTDOT	73	/* RFS specific error */
EBADMSG	74	/* Not a data message */
EOVERFLOW	75	/* Value too large for defined data type */
ENOTUNIQ	76	/* Name not unique on network */
EBADFD	77	/* File descriptor in bad state */
EREMCHG	78	/* Remote address changed */
ELIBACC	79	/* Can not access a needed shared library */
ELIBBAD	80	/* Accessing a corrupted shared library */
ELIBSCN	81	/* .lib section in a.out corrupted */
ELIBMAX	82	/* Attempting to link in too many shared libraries */
ELIBEXEC	83	/* Cannot exec a shared library directly */
EILSEQ	84	/* Illegal byte sequence */
ERESTART	85	/* Interrupted system call should be restarted */
ESTRPIPE	86	/* Streams pipe error */
EUSERS	87	/* Too many users */
ENOTSOCK	88	/* Socket operation on non-socket */
EDESTADDRREQ	89	/* Destination address required */
EMSGSIZE	90	/* Message too long */
EPROTOTYPE	91	/* Protocol wrong type for socket */
ENOPROTOOPT	92	/* Protocol not available */
EPROTONOSUPPORT	93	/* Protocol not supported */
ESOCKTNOSUPPORT	94	/* Socket type not supported */
EOPNOTSUPP	95	/* Operation not supported on transport endpoint */
EPFNOSUPPORT	96	/* Protocol family not supported */
EAFNOSUPPORT	97	/* Address family not supported by protocol */
EADDRINUSE	98	/* Address already in use */
EADDRNOTAVAIL	99	/* Cannot assign requested address */
ENETDOWN	100	/* Network is down */
ENETUNREACH	101	/* Network is unreachable */
ENETRESET	102	/* Network dropped connection because of reset */
ECONNABORTED	103	/* Software caused connection abort */
ECONNRESET	104	/* Connection reset by peer */
ENOBUFS	105	/* No buffer space available */
EISCONN	106	/* Transport endpoint is already connected */
ENOTCONN	107	/* Transport endpoint is not connected */
ESHUTDOWN	108	/* Cannot send after transport endpoint shutdown */
ETOOMANYREFS	109	/* Too many references: cannot splice */
ETIMEDOUT	110	/* Connection timed out */
ECONNREFUSED	111	/* Connection refused */
EHOSTDOWN	112	/* Host is down */
EHOSTUNREACH	113	/* No route to host */
EALREADY	114	/* Operation already in progress */
EINPROGRESS	115	/* Operation now in progress */

ESTALE	116	/* Stale file handle */
EUCLEAN	117	/* Structure needs cleaning */
ENOTNAM	118	/* Not a XENIX named type file */
ENAVAIL	119	/* No XENIX semaphores available */
EISNAM	120	/* Is a named type file */
EREMOTEIO	121	/* Remote I/O error */
EDQUOT	122	/* Quota exceeded */
ENOMEDIUM	123	/* No medium found */
EMEDIUMTYPE	124	/* Wrong medium type */
ECANCELED	125	/* Operation Canceled */
ENOKEY	126	/* Required key not available */
EKEYEXPIRED	127	/* Key has expired */
EKEYREVOKED	128	/* Key has been revoked */
EKEYREJECTED	129	/* Key was rejected by service */
EOWNERDEAD	130	/* Owner died */
ENOTRECOVERABLE	131	/* State not recoverable */
ERFKILL	132	/* Operation not possible due to RF-kill */
EHWPOISON	133	/* Memory page has hardware error */

Linux follows the `0/-E` return convention, i.e, a return value of 0 indicates a successful exit. In case of some error, one is expected to return the negative of the value you want the user space global uninitialised variable, `errno` to be set.

The variable, `errno` is part of a process' uninitialised data segment. When we say that a value is returned to user space, we mean it is actually returned to the `glibc` layer of user space. This layer multiplies the return value by `-1` and sets `errno`. For example, when you are writing a kernel module, the `init_module` system call (any wrapper that is used to load a module into the kernel implements this system call) is expected to return 0 upon success. In case of failure, the kernel emits a warning indicating a non-zero return value (older kernel versions were far more unforgiving and simply unloaded the module!).

Sometimes, instead of returning an integer value, we want to return a pointer. This is accomplished using the `ERR_PTR()` inline function which disguises the pointer as an integer by applying a `void *` cast.



4

Modules

The Linux kernel is monolithic in nature, but that doesn't stop it from having some loadable modules. These modules are pieces of kernel code that can be plugged (unplugged) in (from) the kernel at runtime. Most of the device drivers that work with the kernel are available as loadable modules. This modularity at the OS-level grants us the freedom to customize our system according to our needs.

We will now dive deeper into kernel module programming.

Let us revisit the Hello World example.

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

MODULE_LICENSE("Dual MIT/GPL");
MODULE_AUTHOR("Sukrit Bhatnagar, Purushottam Kulkarni & Aditya Sriram");
MODULE_DESCRIPTION("Customary Hello, World module!");

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

What?! No `stdio.h`?! That's right, we don't need anything from any C library because we are directly interacting with the kernel now. Let us try to understand the code above:

- `linux/init.h` is for the use of `__init` and `__exit` macros.
- `linux/kernel.h` is included to be able to use the `KERN_INFO` macro (which, as we will see, is one of the 8 kernel log levels).
- `linux/module.h` is required by all modules and contains the prototype for the `hello_init` and `hello_exit` functions. This header has to be included by all modules.
- `module_init` is used to specify the entry point of a module, which is given in the form of a function pointer. Although kernel modules are not userspace applications and run with kernel privileges, you

can think of the function specified as being similar to the `main()` function of a C program.

- `module_exit` is used to specify the exit point of a module, which is also given in the form of a function pointer. The function specified is meant to perform cleanup tasks like freeing allocated memory and closing file descriptors before the module is unloaded.
- `printk(KERN_INFO "Hello World!\n")` prints "Hello World!" to the kernel log buffer. It is similar to the `printf` function in C in that it formats a string (in some sense, `printk` is a re-implementation of `printf` in kernel space). However, unlike `printf` which writes to a file (`stdout` by default), `printk` writes to the kernel log buffer in RAM. More on this later!
- `__init` and `__exit` are macros which are used to place the init and cleanup functions in a special code sections (those functions are usually called only once!).

Making both the `init` and `exit` functions `static` ensures they are private to the kernel module. Kernel versions post 2.3.13 have made it so that any function name can be used to describe your `init` and `exit` routines. However, it is considered a good practice to adopt the naming convention where you include the module name in the function.

The kernel also provides a few metadata macros which let you describe your module. These are extremely useful pieces of information that can be viewed by running the `modinfo` command and passing the name of the module. We have listed a few metadata macros below:

- `MODULE_INFO`
- `MODULE_LICENSE`
- `MODULE_AUTHOR`
- `MODULE_DESCRIPTION`
- `MODULE_VERSION`

As an example, try running `modinfo reed_solomon` and observe the output!

Compiling a Module

A simple Makefile for the module you wrote would look something like this:

```
obj-m += <modulename.o>

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

clean:
    make -C /lib/modules/$(uname -r)/build M=$(PWD) clean
```

Assuming you are familiar with the `make` workflow, let us try and decipher exactly what is going on here. The kernel's `Kbuild` system uses two variable strings: `obj-y` and `obj-m`. `obj-y` is a concatenation of all the objects that are to be built into the final kernel image (both, the compressed `bzImage` and uncompressed `vmlinux`). The suffix `y` literally stands for yes i.e, all configurations that were marked `[Y]` during the build process are concatenated into this variable. On the other hand, `obj-m` concatenates all objects that are to be compiled and built as kernel modules (hence the `m`). This is what the first line of the Makefile is for: it tells the `Kbuild` system to compile and build our module, generating the `modulename.o` file which is concatenated to `obj-m`. `all` is the default target for `make` and the rule defining this target does quite a bit:

1. The `-C` option tells the `make` process to perform a change directory (using the `chdir` system call)

to whatever follows it. Running `ls -l` on the path, we can see that the path is a symbolic link to a limited source tree stored in `/usr/src/linux-headers-$(uname -r)`.

2. Inside the source tree, it parses the top-level Makefile.
3. The variable `M` specifies the location where we want our module to be built (in this case, it is the current working directory).

The final target, `clean` is to remove all files generated in the build process and mind you, there are quite a few of them).

Loading/Unloading

If you have reached this step, it's safe to say that you have been able to compile your kernel module successfully and are able to see a `.ko` kernel object file in the destination folder used for compilation. Don't expect anything to happen yet because the module has not been loaded into kernel memory! By default, the user runs in user space and has its own virtual address space. On the other hand, the kernel has its own separate address space and the only way for you, the user, to communicate with the kernel is to rely on system calls. The process by which a module gets loaded into kernel memory is called inserting. There are several utilities to load a kernel module but all use the same system call under the hood, `[f]init_module`. Two commonly used wrappers include `insmod` and `modprobe`. To load your kernel module, run:

```
sudo insmod/modprobe <path_to_module>
```

`modprobe` is an intelligent version of `insmod` and is the recommended option when you want to load modules into the kernel. Usually any module you write will be dependent on other modules. `modprobe` performs a dependency check and loads the module along with its dependencies, which is convenient. However, `modprobe` requires that your module be installed to the default location, `/lib/modules/$(uname -r)`.

Note that you require root privileges to run the above command. This should seem obvious because kernel modules run in the kernel's virtual address space and with elevated privileges. When a process is run as root, it runs with the RUID/EUID equal to the special value zero. You should also know that according to the POSIX capabilities model, a process/thread with the `CAP_SYS_MODULE` capability can load/unload modules. This is a good time to look at the kernel log buffer using `dmesg`. Don't be surprised if you see a message,

```
loading out-of-tree module taints kernel
```

which describes exactly what has happened - you loaded a kernel from outside the source tree. This can be ignored for most cases and is just your Linux distributions way of warning you that it won't be responsible for anything funky that might happen.

Unloaded kernel modules is as simple as loading them - run the `rmmod` utility with root privileges and pass the name of your module:

```
sudo rmmod <modulename>
```

Passing command-line arguments

Let us now look at how one can pass command-line parameters to a kernel module. This can only be done during module insertion (using `insmod`) and the parameters are specified as name-value pairs. For example,

```
sudo insmod <modulename>.ko mp_<parametername>=<parametervalue>
```

(It is a good practice to prefix your module parameter names with `mp_`, which stands for module parameter). The kernel even gives us the ability to pass default values to module parameters!

You might be wondering how exactly is this done. Unlike userspace applications, kernel modules don't have `main()` as their entry point, and hence no `argc`, `argv`! This has to do with the linker and as a kernel programmer, your job is this: declare the module parameter as a static global variable and let the build system know that your variable is to be treated as a module parameter using the `MODULE_PARM()` macro.

The `MODULE_PARM()` macro accepts three arguments: the name of the module parameter, its data type, and permissions (more on permissions later). The kernel also lets you describe the purpose of your module parameter using the `MODULE_PARM_DESC` macro. Running `modinfo` on a module with module parameters displays the list of parameters accepted by your module. You can filter out this information using the `-p` switch. For example, checking the list of module parameters accepted by `drm` returns,

```
$ modinfo -p drm
edid_firmware:Do not probe monitor, use specified EDID blob from built-in data or /lib/
firmware instead. (string)
vblankoffdelay:Delay until vblank irq auto-disable [msecs] (0: never disable, <0: disable
immediately) (int)
timestamp_precision_usec:Max. error on timestamps [usecs] (int)
debug:Enable debug output, where each bit enables a debug category.
    Bit 0 (0x01) will enable CORE messages (drm core code)
    Bit 1 (0x02) will enable DRIVER messages (drm controller code)
    Bit 2 (0x04) will enable KMS messages (modesetting code)
    Bit 3 (0x08) will enable PRIME messages (prime code)
    Bit 4 (0x10) will enable ATOMIC messages (atomic code)
    Bit 5 (0x20) will enable VBL messages (vblank code)
    Bit 7 (0x80) will enable LEASE messages (leasing code)
    Bit 8 (0x100) will enable DP messages (displayport code) (int)
edid_fixup:Minimum number of valid EDID header bytes (0-8, default 6) (int)
```

As you can see, the module accepts five parameters. The name of the parameter is followed by a short description and its datatype is mentioned inside round braces (for example, `timestamp_precision_usec` is an integer while `edid_firmware` is a string).

Can we access/modify parameters after insertion? This depends on the last field passed to `MODULE_PARM()`, which as we said, has to do with permissions/mode. The permissions can be passed as octal numbers or macros that are defined in `include/uapi/linux/stat.h` (although it is recommended that you use the octal format). Whenever you load a module into the kernel, a new directory gets created in the folder `/sys/module`. The `/sys/module` tree consists of the following,

- `/sys/module/<modulename>`: This is always created when the module is loaded dynamically into the kernel.
- `/sys/module/<modulename>/parameters`: This contains files corresponding to each module parameter that can be changed at runtime.

There are other subdirectories like `/sys/module/refcnt` but this will do for now. If the value of the access rights argument is set to 0, the module parameter will not show in the `/sys` directory!

For example, passing `0660` or equivalently, `S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP` gives the read-write permissions to the user and group. In this case, you can examine the contents of the corresponding module parameter file in `/sys/module/<modulename>/parameters` and update the values passed at runtime! Note that there is mechanism by which the kernel knows that the parameter has been updated. We have included the various permissions defined in the kernel below,

```
#define S_IRWXU 00700
#define S_IRUSR 00400
#define S_IWUSR 00200
#define S_IXUSR 00100

#define S_IRWXG 00070
#define S_IRGRP 00040
#define S_IWGRP 00020
#define S_IXGRP 00010

#define S_IRWXO 00007
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001
```

As an example, let us examine the contents of `/sys/module/drm/parameters`,

```
$ ls /sys/module/drm/parameters/
debug edid_firmware edid_fixup timestamp_precision_usec vblankoffdelay
```

which as expected, contains the list of parameters we saw by running `modinfo -p`. We can also check the value taken by each parameter by running `cat` with root privileges. We know that the default value of `edid_fixup` is 6 and can verify this printing the contents of the file corresponding to the parameter,

```
$ sudo cat /sys/module/drm/parameters/edid_fixup
6
```

To find a list of all accepted module parameter datatypes, you can refer to the `include/linux/moduleparam.h` header.

```
/*
...
Standard types are:
*   byte, hexint, short, ushort, int, uint, long, ulong
*   charp: a character pointer
*   bool: a bool, values 0/1, y/n, Y/N.
*   invbool: the above, only sense-reversed (N = true).
*/
```

Auto-loading Modules

So far, we have looked at out-of-tree modules and how they must be inserted into kernel memory by the user using a utility like `insmod`. However, sometimes we might want modules to be automatically loaded at boot. Let us discuss how to do this.

When we were discussing the installation of a custom kernel, we mentioned that kernel object files have to be placed in a `lib/modules/$(uname -r)` for the system to be able to find them and load them into kernel memory. This was achieved by running, `sudo make modules_install`. Therefore, to auto-load out-of-tree modules, all we have to do is add a `modules_install` target in our Makefile! Your new Makefile will look like this,

```
obj-m += <modulename.o>

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

clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean

install:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules_install
```

Running `sudo make install` will install the module to `/lib/modules/$(uname -r)/extra/`. Typically, another utility called `depmod` will be invoked when you run `sudo make install`. The `depmod` utility handles module dependencies and once the module is installed, you can see its effects by running it with the `-dry-run` switch:

```
sudo depmod --dry-run | grep <modulename>
```

We need to do one final thing for our module to be auto-loaded - adding a configuration file in the `/etc/modules-load.d/` directory. Simply create a `<modulename>.conf` file and add the name of your module inside. Alternatively, you can add the name of your module in the `/etc/modules-load.d/modules.conf` file. Note that any line beginning with `#` is treated as a comment. Reboot your system and `lsmod` should show you your module automatically loaded!

How can we assign values to the parameters of a module that has been auto-loaded (remember that a module can be passed parameters at the time of loading)? The easiest way is to create a `modprobe` configuration file in `/etc/modprobe.d/`. As an example, let us take a look at `/etc/modprobe.d/alsa-base.conf`. You should be able to lines like,

```
options bt87x index=-2
options cx88_alsa index=-2
options saa7134_alsa index=-2
```

The syntax is meant to be read as `options <modulename> <parameter-name>=<value>`. Therefore, to pass module parameters, create a `/etc/modprobe.d/modulename.conf` configuration and pass parameters using the `option` lines.

An advantage of installing out-of-tree modules is that you can use `modprobe`, a smarter version of `insmod` to load them. What makes `modprobe` better is that it takes care of module dependencies for you! How does `modprobe` determine the order in which modules are to be loaded, while resolving dependencies? Whenever you install out-of-tree kernel modules to `/lib/modules/$(uname -r)/extra/`, `depmod` generates a `/lib/modules/$(uname -r)/modules.dep` file, which contains dependency information which is used by utilities like `modprobe`.

Some additional information, on modern Linux systems, it is the `systemd` framework that is responsible for auto-loading kernel modules at boot. This is done by the `systemd` service, `systemd-modules-load.service` which parses the contents of files like `/etc/modules-load.d/*`. Sometimes, an auto-loaded kernel module

might misbehave and you would like to prevent it from loading at system boot. This can be done by blacklisting the module. You should be able to see several `*-blacklist.conf` files in the `/etc/modprobe.d` directory.

Exercises

General

- 1 Find out the actual data types underneath the following opaque kernel types:
 - `pid_t`, `uid_t`, `gid_t`, `size_t`, `ptrdiff_t`, `time_t`, `atomic_t`
- 2 Find out the sizes of the following data types on your system:
 - `char`, `bool`, `short`, `int`, `long`, `long long`, `void *`
 - `[u]int8_t`, `[u]int16_t`, `[u]int32_t`, `[u]int64_t`You can guess the sizes of these `"_t"` types without using `sizeof` operator!
- 3 Find out the size of a WORD on your system.
- 4 Find out the total number of system calls available on your system. Also find out what is the limit on the number of arguments you can pass to a system call.
- 5 Find out the limit on the number of characters allowed in a line of the syslog buffer.

Modules

- 1 Write a module to print your name and email ID in the kernel logs. Name your init and cleanup functions as "hi" and "bye", respectively, instead of `init_module` and `cleanup_module`. You will find `module_init` and `module_exit` macros useful here.

```
[must do] (module) {general/hello.c}
```
- 2 Write a module to try out various kernel logging levels using `printk`. Print a string to the *terminal* using one of the kernel log levels. Also find out all the log levels that print to the terminal by default. (NOTE: no log messages will be printed on pseudoterminals (pts), so use one of the ttys to try it out!).

```
[must do] (module) {general/log.c}
```
- 3 Write a module to crash the kernel in the easiest way you can imagine! Verify by looking at the kernel logs for a kernel oops message. Now reboot!

```
[must do] (module) {general/crash.c}
```
- 4 There is a symbol `LINUX_VERSION_CODE` in `include/generated/uapi/linux/version.h` which represents the code for current kernel version. For a kernel version `a.b.c`, this code is calculated as:
`a * (2^16) + b * (2^8) + c`. Write a module to calculate and print the kernel version, by using and decoding this code. (HINT: some bit manipulation skills required!).

```
[can do] (module) {general/metadata.c}
```

- 5 Using the `min3` and `max3` macros defined in `include/linux/kernel.h`, write a module that prints the minimum and maximum value among three signed integers taken in from command line.

```
[can do] (module) {general/minmax3.c}
```

- 6 Write a module to accept an array of integers as the command-line argument. The number of elements should be fixed (say 10) and the module should exit after printing suitable message if the number of elements passed is not equal to 10; print the integers in the array otherwise. (HINT: take a look at `module_param_array()`).

```
[must do] (module) {general/array.c}
```

- 7 Write another module to extend the module you have written for the exercise 6 above. Sort the integers in decreasing order and print them. Use the `sort` function in `include/linux/sort.h`. (HINT: you might want to define your own compare function).

```
[can do] (module) {general/sort.c}
```

- 8 Write a module to create and manipulate a kernel linked list. Use the struct definition given below for the linked list node. Your module should take a number from the command-line which will specify the number of nodes to add in the linked list.

For each node, the member `num` should be a random number generated using an inbuilt kernel function, and the member `sqrt` should contain the square root of `num` (decimal part not needed). Add the nodes to the end of the list and then print the members of the nodes in the linked list in reverse, and then delete and free all the nodes in the list. Use the functions `get_random_long` and `int_sqrt` in `linux/random.h` and `linux/kernel.h`, respectively. (HINT: You might want to take a look at `list_add_tail` and `list_for_each_entry_safe_reverse`).

```
struct info
{
    unsigned long num;
    unsigned long sqrt;
    struct list_head list;
};
```

```
[must do] (module) {general/list.c}
```

- 9 Extend the module you wrote in the exercise above to measure the "average" time taken to insert a new node in the linked list. Use `getnstimeofday` to get the time before and after the insert operation in nanoseconds.

```
[must do] (module) {general/elaptime.c}
```

- 10 Load the module you wrote in Exercise 1 (remove the `__init` macro from the function signature before you make it!). Then write a kernel module which accepts the name of that kernel module as command-line argument and traverse the list of loaded kernel modules to find it. When you find it, print its name as well as call its "init" function! Refer to the `struct module` definition in `linux/module.h`.

```
[can do] (module) {general/search_mod.c}
```

- 11 Write a simple module and make it persistent, i.e. it should automatically load on system boot. Reboot and check if it is loaded using `lsmod`. (HINT: take a look inside `/lib/modules`).

- 12** Write a module to print the current date and time. Use `do_gettimeofday` for fetching the current time in seconds and the function `time_to_tm` which populates a `struct tm` based on that value. Keep the GMT time zones in mind while you are at it!

```
[must do] (module) {general/datetime.c}
```

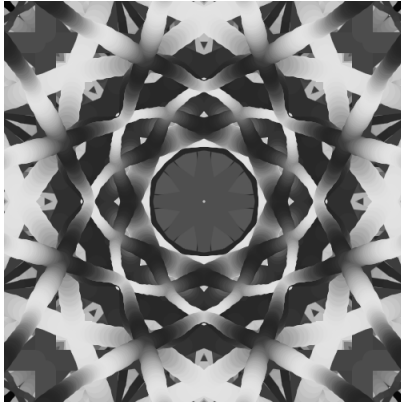
- 13** Write a module to access a take a command-line argument which sets the value of a variable. Then access this variable from another kernel module. Use `EXPORT_SYMBOL` in the first module and access it in another module by using `extern`. (HINT: The second module should be loaded only after first one is loaded, and similarly the first one should be unloaded only after second one is unloaded).

```
[must do] (module) {general/export.c,export_2.c}
```

- 14** Write a kernel module to print the contents of all the x86_64 registers that the kernel uses, namely cr0, cr2, cr3, cr4 and cr8. Using the value returned from cr4, check if PAE (Physical Address Extension) is enabled on your machine.

```
[must do] (module) {general/cr.c}
```

- 15** Write a module to get a pointer to the task that is currently executing on the current CPU i.e. calculate the value of `current` macro.



5

Design Patterns

Data Types

Refer: `include/linux/moduleparam.h` The following data types are supported as command line arguments to a kernel module:

byte (unsigned char)	short	ushort (unsigned short)
int	uint (unsigned int)	long
ulong (unsigned long)	ulong (unsigned long long)	charp (char *)

Data Structures

Lists

The kernel has an unconventional, but efficient implementation of a linked list which is generic in nature. Normally, a linked list node consists of data and pointers to the next and/or previous nodes. However, the implementation of the linked list used in the kernel only contains pointers to the next and/or previous nodes! This means that the linked list is completely oblivious to the type of structure being linked, lending the implementation its generic nature. Such linked lists are referred to as `intrusive`, which simply means that the linked list elements are not structures themselves, but rather each node of the linked list is embedded into the structure which is supposed to be linked. There are broadly two different types of intrusive-headed-doubly linked lists that the kernel provides natively, as mentioned below.

Some more terminology, **headed** means that a special node, usually called head, will act as the pointer to the first (and in some cases, even the last) node in the list. **Double** simply means that from a node in the list, you can traverse to the next as well as the previous elements, provided they exist.

Circular doubly linked list

The Linux kernel includes an implementation for a circular doubly linked list, which can be found in `include/linux/types.h:list_head`.

```
struct list_head {
    struct list_head *next, *prev;
};
```

As you can see, each node in the list has a pointer to the next as well as the previous element, but does not store the data. If the list has only one element, its `next` and `prev` will point to itself (hence the circular nature). A head node is used to identify a list, hence the `struct` name `list_head`. The type of head node is the same as other data nodes, but its `next` points to the first element in the list and its `prev` points to the last node. Note that you don't really need a head node for the linked list to function. The kernel includes several routines to manipulate the data structure which can be found in `include/linux/list.h`. The implementation is simple enough and hence, has been inlined. Access methods including `list_entry`, which is wrapper for `container_of`, allow you to access the structure being referenced by the `list_head` pointer. This linked list implementation was motivated by the fact that sometimes, you want to be able to traverse the list, both forwards and backwards which is why the `prev` pointer was included. This has the added advantage of being able to access the tail in $O(1)$ time. We have included API commonly used with lists below, for the reader's convenience.

Hash lists

```
struct hlist_head {
    struct hlist_node *first;
};
struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

This implementation of a linked list is useful in applications where it is not necessary to be able to access the last entry in constant time and might benefit from the space reduction due to not storing a `prev` pointer. This turns out to be extremely useful in hash tables where all the nodes are stored in an array. These lists have separate types for the head node and the data nodes since you need to be able to access the head of such a list. Just like `list_head`, helper methods for this linked list are defined in `include/linux/list.h`. As an example, we can look at the use of `hlist_head` to create a hash table defined in `include/linux/hashtable.h`.

Why has a double pointer been used in place of the normal `prev` pointer? This is because the `hlist` is implemented using two structures, `hlist_head` and `hlist_node`. This separation exposes an interface to the head of the list but also causes an issue. What will `prev` point to? The head of the list is of type `hlist_head`, while every other element in the list is of type `hlist_node`. Using a double pointer fixes this problem because the two structures share a common interface, which is a pointer to the type `hlist_node`.

Red-black Trees

```
struct rb_node {
    unsigned long __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
/* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root {
    struct rb_node *rb_node;
};
```

The kernel also includes an implementation of red-black trees in `include/linux/rbtree.h` and `lib/rbtree.c`. We won't go into the internals of a red-black tree, but just know that it is a semi-balanced binary search tree with operations like insert, delete and search taking $O(\lg n)$ time. Just like the circular doubly linked list, the tree node is embedded in the structure for which the tree is being constructed. What's interesting is that the kernel does not include code for insert or search functions, and these must be implemented by the user themselves! The reason behind this is performance. To implement a search function, you would need to pass a comparison function using a function pointer. Since comparison functions are simple, the cost of following the pointer and making the function call would be dominate. In a similar spirit, there is no code for insertion. The user is expected to write code for the search, if it fails, find the position where the new node should be inserted, insert the node and balance the tree. You will find functions for to link the new node and rebalance the tree. Finally, at the root of every red-black tree, is a `rb_root` structure.

Radix Trees

Besides red-black trees, the kernel includes a second kind of tree - radix trees. However, the way radix trees are used in the kernel is more general than you might expect. Radix trees can be found in two locations, `include/linux/radix-tree.h`, `lib/radix-tree.c` and `include/linux/idr.h`, `lib/idr.c`. A Linux radix tree lets you associate a pointer with a very long integer key. Radix trees have some features that are also motivated by kernel-specific needs. Each node in the radix tree consists of a number of slots, which is quite large. The integer key can be used to index into tree with portions of the key being used to index into node slots. Unlike lists and red-black trees, a radix tree is not embedded inside the data structure which means that memory might have to be allocated to add items to the data structure.

Kernel Log Buffer

If you ran the `hello` module described in the warming up section, you would have noticed that the `printk` kernel API is very similar to the `printf` method that you know from C. This is not an accident! It is extremely important for you to understand that there is no concept of a "library" in kernel space (the `lib/` folder in the kernel source tree is the closest you will ever get to a library in the kernel). This means that you do not have access to the `printf` API for debugging purposes. Instead, kernel contributors decided to re-implement the API within the kernel and the code for the same can be found in the `kernel/printk` folder of the source tree.

Unlike `printf`, `printk` does not write to terminal window (internally, `printf` invoke the `write` syscall which by default, writes to `stdout` which is the terminal window in most cases) and has three possible write destinations:

1. The kernel log buffer in RAM (volatile)
2. A log file on the disk (non-volatile)
3. The console

This is also why we did not see anything get printed to the console in the example and had to use the `dmesg` command to view the "Hello, world!" string.

You would have also noticed the macro, `KERN_INFO` before the string, "Hello, world!". There is no comma between the macro and string which means that this is not a parameter but rather one of the 8 kernel log levels. The 8 kernel log levels are defined in the file, `include/linux/kern_levels.h`

```
#define KERN_SOH          "\001"          /* ASCII Start Of Header */
#define KERN_SOH_ASCII   '\001'

#define KERN_EMERG        KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT        KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT         KERN_SOH "2"    /* critical conditions */
#define KERN_ERR          KERN_SOH "3"    /* error conditions */
#define KERN_WARNING      KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE       KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO         KERN_SOH "6"    /* informational */
#define KERN_DEBUG        KERN_SOH "7"    /* debug-level messages */

#define KERN_DEFAULT      ""              /* the default kernel loglevel */
```

`KERN_SOH` is the kernel start of header value and is appended to the beginning of all log levels, which you can see are nothing but numbers. Note that these log levels are actually representing control strings for the console. For instance, `KERN_INFO` would actually correspond to `"\0016"`. If you do not specify the kernel log level, it defaults to `KERN_WARNING`. But, it is considered a good practice to mention a suitable log level. This might get tedious but can be avoided using the `pr_` macros which are defined in `include/linux/printk.h`.

printk

```
struct printk_info {
    u64    seq;                /* sequence number */
    u64    ts_nsec;           /* timestamp in nanoseconds */
    u16    text_len;          /* length of text message */
    u8     facility;          /* syslog facility */
    u8     flags;              /* internal record flags */
    u8     level;              /* syslog level */
    u32    caller_id;         /* thread id or processor id */

    struct dev_printk_info dev_info;
};
```

When you call `printk`, an object of type `printk_log` will be generated and it will be added to a log buffer. A global variable `log_buf` is an array of pointers, where each element is a "line" you see in the syslog. This is the kernel's implementation of the syslog buffer. The function `log_store` does the main work in adding a `printk_log` to the `__log_buf`.

`__log_buf` is implemented as a ring buffer and by default, is quite small! The size is controlled by the kernel configuration symbol, `CONFIG_LOG_BUF_SHIFT` which defines two's exponent. By default, the symbol is set to 18 so the size of the buffer is 256 KB. Therefore, too many print statements will fill the entire buffer and overwrite valuable log data. Thankfully, your distribution will typically write to a file (which is non-volatile). Debian based systems store the file in `/var/log/syslog`, which is maintained by the `systemd` framework. Logging is performed using the `systemd-journal` daemon while `journalctl` allows you to query the journal. Note that the daemon logs messages from almost everything possible - kernel, applications, drivers.

We have seen how `printk` writes to the kernel log buffer and a log file on the disk. Let's finally see how we can get `printk` to display messages on the console window. It turns out that messages with a low enough log level (high in priority) will be displayed on the console window, in addition to log buffer! The kernel uses the `proc` filesystem to decide which log levels are "important" enough to be displayed on the console. This can be examined by running,

```
$ cat /proc/sys/kernel/printk
4 4 1 7
```

The four numbers displayed are log levels and the particular sequence means the following:

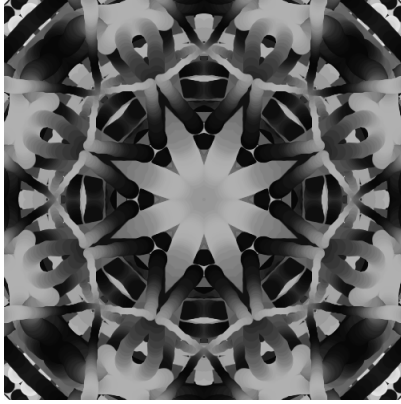
- The first number is the current console log level. It is 4 by default which corresponds to `KERN_WARNING`. All `printk` instances with log level less than 4 will be printed to the console device.
- The second number is the default log level for messages explicitly lacking one (say you forgot to add a macro to your `printk` statement)
- The third number is the minimum log level allowed
- The fourth number is the default log level at boot-time

By appropriately modifying the numbers in the above file, you can control what messages get logged to the console window which is a very useful ability.

Finally, you can also generate kernel messages from userspace! This can be done by directly writing to the `/dev/kmsg` character device file, which provides userspace access to the kernel's log buffer:

```
sudo bash -c "echo hello! > /dev/kmsg"
```

Running `dmesg | tail` should show you the message, "hello!" appended to the log buffer.



6

Processes

Suggested readings

Books

[lkd]: 3, 4

[ulk]: 3, 7, 9

Others

<https://tampub.uta.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf>

<http://sunnyeves.blogspot.com/2010/09/sneak-peek-into-linux-kernel-chapter-2.html>

<https://www.win.tue.nl/~aeb/linux/lk/lk-10.html>

Introduction

Let us start with processes, or as Linux calls them, tasks! They are the ones who actually get the work done. Note that task has a more general meaning inside the kernel, and we may use this term to refer to any process or kernel thread. Basically, anything that can be independently scheduled by the kernel is a task. Now, even though a thread is just an execution path within a process, it turns out that threads are more important to the kernel than processes. This is because in the Linux kernel, threads are **kernel schedulable entities** (KSEs) and every thread (userspace or kernel) gets mapped to a kernel metadata structure, called the task structure. The task structure (also known as a process descriptor, which is a source of much confusion) contains everything that the kernel needs to know about the thread, including information on memory, credentials, namespaces, IPC structures, etc. Some of these attributes will be inherited by a child process or thread (paging tables, namespaces to name a few), while others will not be inherited, but reset. You might be wondering, does the kernel know if a task is a process or thread? The answer is yes, and we will see how soon enough. To conclude, every process consists of one or more threads, which in turn map to the task structures in the kernel.

Note: It should be pointed out threads of the same process share the process virtual address space (VAS), except the stack, which is private to each thread. Additionally, the number of stacks that a thread requires

is related to the number of privilege levels supported by the CPU. Modern OSes like Linux support two privilege levels, and hence, each thread has two stacks - a userspace stack and a kernel stack. The userspace stack comes into play when the thread executes userspace code, while the kernel stack comes into play when you execute kernel code in process context. This rule has one exception - kernel threads. Kernel threads only have one stack because they cannot see userspace.

Structs and Flags

The Process Control Block (PCB) is represented in the kernel by the structure `task_struct` located in `include/linux/sched.h`. It looks something like the code below; well, a bit more cryptic than this to be frank. This code includes only the process attributes relevant to us right now, and there are around a couple hundred such attributes which it encapsulates! Nevertheless, the order of member declarations is preserved here for better correspondence.

```
struct task_struct {
    ...
    struct thread_info      thread_info;
    volatile long           state;
    void                    *stack;
    unsigned int            flags;
    unsigned int            cpu;

    int                     prio, static_prio, normal_prio;
    const struct sched_class *sched_class;
    struct sched_entity     se;
    unsigned int            policy;
    struct sched_info       sched_info;

    struct list_head        tasks;

    struct mm_struct        *mm, *active_mm;

    int                     exit_state, exit_code, exit_signal;

    pid_t                   pid, tgid;

    struct task_struct      *parent;
    struct list_head        children, sibling;
    struct task_struct      *group_leader;

    struct pid_link         pids[PIDTYPE_MAX];
    struct list_head        thread_group, thread_node;

    u64                     utime, stime, gtime;

    unsigned long           nvcsw, nivcsw;
    u64                     start_time;
    unsigned long           min_flt, maj_flt;

    const struct cred        *cred;

    char                    comm[TASK_COMM_LEN];

    struct nameidata        *nameidata;
    struct fs_struct        *fs;
```

```

struct files_struct      *files;

struct signal_struct     *signal;
struct sighand_struct    *sighand;
sigset_t                 blocked;

spinlock_t               alloc_lock;

struct task_io_accounting ioac;

int                       nr_dirtied;

atomic_t                 stack_refcount;

struct thread_struct     thread;
...
};

```

- `thread_info`

Pointer to the corresponding `thread_info`. The implementation of the `thread_info` is architecture-specific. It is needed as the first member in our struct so that thread info can be easily extracted by casting a `task_struct` pointer to a `thread_info` pointer.

Refer [arch/x86/include/asm/thread_info.h:56](#) .

- `state`

Process state. Value `-1` represents that the process is not runnable, `0` means it is runnable (state `TASK_RUNNING`), and any positive value (`>0`) means it is stopped. All the possible state values are defined in `include/linux/sched.h` and are used as bitmasks to make up a single `long` variable. Here are the values for reference:

```

/* Used in tsk->state: */
#define TASK_RUNNING      0x00000000
#define TASK_INTERRUPTIBLE 0x00000001
#define TASK_UNINTERRUPTIBLE 0x00000002
#define __TASK_STOPPED    0x00000004
#define __TASK_TRACED     0x00000008
/* Used in tsk->exit_state: */
#define EXIT_DEAD         0x00000010
#define EXIT_ZOMBIE       0x00000020
#define EXIT_TRACE        (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED       0x00000040
#define TASK_DEAD         0x00000080
#define TASK_WAKEKILL     0x00000100
#define TASK_WAKING       0x00000200
#define TASK_NOLOAD       0x00000400
#define TASK_NEW          0x00000800
#define TASK_RTLOCK_WAIT  0x00001000
#define TASK_FREEZABLE    0x00002000
#define __TASK_FREEZABLE_UNSAFE (0x00004000 * IS_ENABLED(CONFIG_LOCKDEP))
#define TASK_FROZEN       0x00008000
#define TASK_STATE_MAX    0x00010000

```

- `stack`

Pointer to the kernel stack associated with the process. The user stack is provided as a virtual memory area (vma) to the process.

- `flags`

Process flags. All the possible flag values are defined in `include/linux/sched.h` and are used as bit masks to make up a single `long` variable.

```
PF_IDLE          0x00000002 /* I am an IDLE thread */
PF_EXITING       0x00000004 /* Getting shut down */
PF_EXITPIDONE    0x00000008 /* PI exit done on shut down */
PF_VCPU          0x00000010 /* I'm a virtual CPU */
PF_WQ_WORKER     0x00000020 /* I'm a workqueue worker */
PF_FORKNOEXEC    0x00000040 /* Forked but didn't exec */
PF_MCE_PROCESS   0x00000080 /* Process policy on mce errors */
PF_SUPERPRIV     0x00000100 /* Used super-user privileges */
PF_DUMPCORE      0x00000200 /* Dumped core */
PF_SIGNALED      0x00000400 /* Killed by a signal */
PF_MEMALLOC      0x00000800 /* Allocating memory */
PF_NPROC_EXCEEDED 0x00001000 /* set_user() noticed that RLIMIT_NPROC was exceeded */
PF_USED_MATH     0x00002000 /* If unset the fpu must be initialized before use */
PF_USED_ASYNC    0x00004000 /* Used async_schedule*(), used by module init */
PF_NOFREEZE      0x00008000 /* This thread should not be frozen */
PF_FROZEN        0x00010000 /* Frozen for system suspend */
PF_KSWAPD        0x00020000 /* I am kswapd */
PF_MEMALLOC_NOFS 0x00040000 /* All allocation requests will inherit GFP_NOFS */
PF_MEMALLOC_NOIO 0x00080000 /* All allocation requests will inherit GFP_NOIO */
PF_LESS_THROTTLE 0x00100000 /* Throttle me less: I clean memory */
PF_KTHREAD       0x00200000 /* I am a kernel thread */
PF_RANDOMIZE     0x00400000 /* Randomize virtual address space */
PF_SWAPWRITE     0x00800000 /* Allowed to write to swap */
PF_NO_SETAFFINITY 0x04000000 /* Userland is not allowed to meddle with cpus_allowed */
/*
PF_MCE_EARLY     0x08000000 /* Early kill for mce process policy */
PF_MUTEX_TESTER  0x20000000 /* Thread belongs to the rt mutex tester */
PF_FREEZER_SKIP  0x40000000 /* Freezer should not count it as freezable */
PF_SUSPEND_TASK  0x80000000 /* This thread called freeze_processes() and should not
    be frozen */
```

- `cpu`

CPU number on which the process is currently executing. This value is transient and changes when the process is scheduled on a different CPU.

- `prio`, `static_prio`, `normal_prio`

Priorities associated with the process. `static_prio` is the value assigned to the process when it is created. This value is computed from the user "nice" values using `NICE_TO_PRIO`. `normal_prio` is computed according to the scheduling policy used by the process. Child processes inherit this priority from their parents upon fork. `prio` is one taken into account by the scheduler and changes dynamically. There is another field `rt_priority` for real-time tasks.

- `sched_class`

Scheduling class associated with the process. `struct sched_class` represents a NULL-terminated singly linked list, where each struct element contains some function pointers. These function pointers are initialized with the appropriate function logic when a scheduling class struct is created. The scheduling class structure represents an "interface" for each scheduling class. There are 5 schedul-

ing classes defined in the kernel scheduler: `stop_sched_class`, `dl_sched_class`, `rt_sched_class`, `fair_sched_class`, `idle_sched_class`.

Refer [kernel/sched/sched.h:1463](#) and [kernel/sched/core.c](#).

- `se`

Scheduling entity. The scheduler does not work with processes directly; instead it takes a set of scheduling entities. Threads, thread groups, processes, process groups; all are scheduling entities. `struct sched_entity` contains information useful in making scheduling decisions; "weight", execution start time, total execution time and number of migrations, among other properties.

Refer [include/linux/sched.h:445](#).

- `policy`

Scheduling policy to be used for this process. The following values can be assigned to this field:

```
SCHED_NORMAL    0
SCHED_FIFO      1
SCHED_RR        2
SCHED_BATCH     3
/* SCHED_ISO    */
SCHED_IDLE      5
SCHED_DEADLINE  6
```

- `sched_info`

Miscellaneous counters related to scheduling. The members of this struct are available only if `CONFIG_SCHED_INFO` is enabled in the kernel.

Refer [include/linux/sched.h:281](#).

- `tasks`

(Doubly linked) list of all the tasks in the system. Useful if you need to traverse through all the processes in the system. `for_each_process` uses the same.

- `mm`, `active_mm`

The "memory" of a process. Each process will have a `mm` associated with it. For a process executing in user mode, both `mm` and `active_mm` will point to the same memory struct. When a process switches to kernel mode, or for kernel processes/threads, the `active_mm` will point to the memory struct of the previously scheduled process on the same CPU and `mm` will be NULL. The `mm_struct` contains some important fields related to memory management such as address of the page directory, list of all virtual memory areas (vma) and the start and end addresses of various sections (code, data, heap, stack). This will be discussed with further depth in the chapter dedicated to memory management.

Refer [include/linux/mm_types.h:352](#).

- `exit_state`, `exit_code`, `exit_signal`

When a process is currently executing, `exit_state` can have any of the values that `state` can have. When the process is exiting, or waiting to be reaped, it can have the following values:

EXIT_DEAD	0x0010
EXIT_ZOMBIE	0x0020
EXIT_TRACE	(EXIT_ZOMBIE EXIT_DEAD)

The field `exit_code` can have two sets of values. If the process exited due to a signal, this will contain the corresponding signal number. If the process called `exit` explicitly and passed `x` as its argument, it will have a value `(x & 0xff) << 8`. The field `exit_signal` usually contains the value 17 (`SIGCHLD`).

- `pid`, `tgid`

Process id `pid` is (supposedly) the unique identifier for a process in the system; two processes cannot have the same pid (unless they belong to different namespaces). But, the kernel has a different notion of pid altogether! `pid` is more like tid (thread ID) and is unique to a process OR a (kernel) thread. (Recall that user-level threads such as pthreads are each mapped to a kernel thread). Then what will be the `tgid`. It is the pid of the group leader, which may be the process that created all the threads. Note that the concept of group leader applies only to a process and the threads it spawns. If a process forks a child, then that child will be the group leader of its own group and will not share the thread group of its parent.

Ironically, `getpid` returns the `pid` of the group leader, which is `tgid`, and `gettid` returns the `pid`. For a process, `getpid` and `gettid` will be the same. If the process spawns some pthreads, all of the pthreads will have a different kernel-space pid (`pid`) but the same user-space pid (`group_leader->pid` or `tgid`).

Refer [kernel/sys.c:882](#) .

- `parent`

Parent, or better the "effective" parent. This is the process that is supposed to reap the process (wait for it) and receives `SIGCHLD` when child terminates. There is also a `real_parent` that is the process which forked, or to be more precise, cloned this process. Unless someone is ptracing, these two parents are the same.

- `children`, `sibling`

Children of the process i.e. all the processes cloned from it. `children` acts as the head node for the list. `children.next` will point to the list element of the first child and `children.prev` will point to the list element of the last child in the list. `sibling` is the linked list node embedded in each sibling's `task_struct`.

- `group_leader`

Leader of the thread group this task belongs to. Every process is its own group leader. But if a process has spawned multiple threads, all of those threads will have that process as their group leader. And as we know by now, the `pid` of the group leader will be the `tgid` of all the threads (it spawned). All of those threads as well as the process are said to be in the same "thread group". NOTE: all the threads and processes in a group will have the same `parent`, i.e., even if the process had spawned those threads, the parent of that process will also be the parent of all these threads.

- `pids`

Hash table linkage of the several types of pids a process can have. There are briefly three types of such pids: `PIDTYPE_PID`, `PIDTYPE_PGID` (process group ID) and `PIDTYPE_SID` (session ID). These

types are declared inside an enum named `pid_type` and act as buckets in the hash table.

A session contains a number of process groups, and a process group contains a number of processes, and a process contains a number of threads. Typically, there is a session corresponding to every tty in use. When you open the terminal, it starts a shell (such as bash) on a tty device and this becomes a new session.

We already know about thread group ID (tgid) which is the `PIDTYPE_PID` of the group leader. Similarly, process group ID of a process is the `PIDTYPE_PGID` of its group leader and its session group ID is the `PIDTYPE_SID` of the group leader. The hash table is implemented by the kernel using linked lists, with chaining for handling collisions.

- `thread_group`, `thread_node`

They both represent a linked list of all the processes in a thread group. The only difference is that `thread_group` has its head inside the `task_struct` of group leader, whereas `thread_node` has its head inside the `signal` shared by all the members in a thread group. Using `thread_node`, you can iterate all the members of a thread group, including the group leader. But using `thread_group`, you can traverse through all members except the leader as it has been used up for being the list's head. `for_each_thread` macro uses `thread_node` internally.

- `utime`, `stime`, `gtime`

`utime` is the time spent by the process in user mode. `stime` is the time spent in kernel mode (usually through system call invocations). `gtime` is the guest time. All these times are in the unit of clock ticks rather than seconds. To convert them back to seconds, you have to divide these values by `CLK_TCK` configuration parameter (`getconf CLK_TCK`).

- `nvcs`, `nivcs`

`nvcs` and `nivcs` count the number of context switches that the current task has undergone. `nvcs` counts the number of voluntary context switches, while `nivcs` counts the number of involuntary context switches. These variables are exposed to userspace via the `taskstats` structure, which maintains per-process and per-task stats that are returned when the last task of the process exits.

- `start_time`

- `min_flt`, `maj_flt`

- `cred`

- `comm`

- `nameidata`

- `fs`, `files`

- `signal`

- `sighand`

- `blocked`

You must be wondering why the `task_struct` fields are so arbitrarily arranged declared the kernel. Think in terms of caches!

Now that we have looked at some of the members of the task structure, let us try and understand how the task structures are organised in the kernel. It turns out that all the task structure objects in kernel memory and placed in a circular doubly linked list, called the task list. This is one of the most important uses of the kernel data structure we saw in the last chapter. At any given time, there are several processes/threads alive on the system. How do we find the task structure corresponding to the thread/process that is currently running kernel code in process context? The kernel developers came up with a scheme that uses the `current` macro. The `current` pointer can be followed to the `task_struct` of the thread that is currently running kernel code.

The implementation of the `current` macro is extremely architecture-specific and can be found in the file, `arch/<arch>/asm/include/current.h`. For example, in the PowerPC architecture, `current` is stored in register `r2` for speed. x86 uses a per-CPU variable to hold the value of `current`.

There is a small caveat to using `current` in your code - you can only use the macro in process context. In general, there are two ways by which you can run kernel code - process or interrupt context. Kernel code can be run in process context by invoking a system call, which transfers control over to the OS that in turn, runs with elevated privileges. This can also happen when the processor throws an exception. Process context allows you to run switch to kernel mode in an synchronous manner. On the other hand, in interrupt context, a hardware interrupt causes the CPU to stop doing whatever it is doing and switch to kernel mode to execute an interrupt service routine (ISR). This takes place asynchronously.

Therefore, before using `current`, you must determine the context within which you are executing kernel code. This can be done using the `in_task()` macro defined in `include/linux/preempt.h`. `in_task()` return `True` if your code is executing in process (or task context) and return `False` if your code is running in interrupt/atomic context.

One more thing: `current` gives you access to the `task_struct` of the process/thread currently running kernel code and you can use it to access attributes like PID, PPID, etc. However, rather than doing this directly, it is recommended that you use helper methods defined in `include/linux/sched.h` because of a lock that must be taken.

Finally, the kernel provides APIs defined in `include/linux/sched/signal.h` using which you can iterate through the task list. You can iterate through the task list displaying,

- All processes currently alive on the system. This can be done using the `for_each_process` macro,

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

The `init_task` macro points to the very first userspace process, `systemd` (initially known as `init`). Note that this allows you to iterate over the `main()` thread of every active process.

- All threads currently alive on the system. This can be done using the pair of `do_each_thread()` and `while_each_thread()` macros,

```
#define do_each_thread(g, t) \
    for (g = t = &init_task ; (g = t = next_task(g)) != &init_task ; ) do

#define while_each_thread(g, t) \
```

```
while ((t = next_thread(t)) != g)
```

These signal values are defined in `uapi/asm-generic/signal.h`.

```
SIGHUP      1  /* Hangup detected on controlling terminal or death of controlling process */
SIGINT      2  /* Interrupt from keyboard */
SIGQUIT     3  /* Quit from keyboard */
SIGILL      4  /* Illegal Instruction */
SIGTRAP     5  /* Trace/breakpoint trap */
SIGABRT     6  /* Abort signal from abort(3) */
SIGIOT      6  /*
SIGBUS      7  /* Bus error (bad memory access) */
SIGFPE      8  /* Floating point exception */
SIGKILL     9  /* Kill signal */
SIGUSR1    10  /* User-defined signal 1 */
SIGSEGV    11  /* Invalid memory reference */
SIGUSR2    12  /* User-defined signal 2 */
SIGPIPE    13  /* Broken pipe: write to pipe with no readers */
SIGALRM    14  /* Timer signal from alarm(2) */
SIGTERM    15  /* Termination signal */
SIGSTKFLT  16  /*
SIGCHLD    17  /* Child stopped or terminated */
SIGCONT    18  /* Continue if stopped */
SIGSTOP    19  /* Stop process */
SIGTSTP    20  /* Stop typed at terminal */
SIGTTIN    21  /* Terminal input for background process */
SIGTTOU    22  /* Terminal output for background process */
SIGURG     23  /* Urgent condition on socket */
SIGXCPU    24  /* CPU time limit exceeded */
SIGXFSZ    25  /* File size limit exceeded */
SIGVTALRM  26  /* Virtual alarm clock */
SIGPROF    27  /* Profiling timer expired */
SIGWINCH   28  /*
SIGIO      29  /* Pollable event */
SIGPOLL    29  /* Pollable event */
SIGPWR     30  /*
SIGSYS     31  /* Bad argument to routine */
SIGUNUSED  31  /*
SIGRTMIN   32  /*
SIGRTMAX   64  /*
```

List of important functions and macros

Warning: Importance is relative.

Exercises

1 Write a module to print the following information about the kernel internals:

- Maximum and minimum values for the pid of a user process (verify from the output of `dmesg | grep pid_max`)

- The range of reserved pids i.e. those pids which can never be assigned to a user process.

```
[must do] (module) {process/proc_values.c}
```

- 2 Write a module to print the pid and name of the "current" process. Reason about the output that you see.

```
[must do] (module) {process/current.c}
```

- 3 Write a module to know the names of processes with pids 0, 1 and 2. Use `for_each_process` macro defined in `linux/sched/signal.h` for traversing all the processes. (NOTE: you might have to put some extra effort to print the name of process with pid 0!).

```
[must do] (module) {process/three.c}
```

- 4 Write a module to print all ancestors, siblings and children of a process whose pid is passed as a command-line argument to the module. Refer to `struct task_struct` in `linux/sched.h`.

```
[must do] (module,list) {process/family.c}
```

- 5 Write a module to print the name of a process whose pid is passed as a command-line argument to the module. Use the functions `pid_task` and `find_get_pid` defined in `linux/pid.h` to lookup the `task_struct` corresponding to the pid.

```
[must do] (module) {process/pid.c}
```

- 6 Write a `ps`-like kernel module to print information about all process in the system. e.g., the following fields: pid, parent pid, name, user time, sys time, priority, uid, gid, current cpu, flags, number of open files, policy, state and terminal (if used). You should make use of `for_each_process` macro defined in `linux/sched/signal.h`.

```
[must do] (module) {process/ps.c}
```

- 7 Write a module to rename a process whose pid is passed as a command-line argument. Make use of `comm` field in `task_struct` and verify the change by looking at `/proc/<pid>/comm`. You will need to hold a lock on the struct while doing the renaming!

```
[must do] (module,proc) {process/rename.c}
```

- 8 Write a module to implement `pstree`-like functionality. The module should take in the pid of a process as a command-line argument and must print all its descendants in a similar fashion. (HINT: does depth-first search ring any bells?).

```
[must do] (module) {process/pstree.c}
```

- 9 Write a program in C to fork processes in a tree-like structure. Your program should first fork two processes. Then each of those two children must fork two children, and so on. The program should be generic in the sense that the level of this fork-tree must be adjustable. The last children forked (or the leaf of this fork-tree) must print a single unique element from a global array, and all elements of the array should be printed due to execution of all last-level children. (HINT: use recursion to fork processes!). Run the `pstree`-like module you wrote in the previous exercise on the process to verify the fork-tree.

```
[can do] (fork) {process/tree-fork.c}
```

- 10** Write a module to kill a process! The pid of the process to kill is passed as a command-line argument and SIGKILL must be delivered to it by the kernel itself. (HINT: you might want to take a look at `send_sig()` in `linux/sched/signal.h`).
- ```
[can do] (module) {process/sigkill.c}
```
- 11** Write a module to create a kernel thread. Make the thread sleep for 20 seconds when it runs. Take a look at `kthread_run` and `.`. Run `ps aux | grep 'thread name'` to view it. Try sending SIGKILL to it using the method described in the previous question. Explain the unexpected behaviour observed.
- ```
[can do] (module) {process/kthread.c}
```
- 12** Write a module to create a kernel thread (using `kthread_create`) which does some "busy" waiting for 20 seconds. Use `getnstimeofday64` to check the seconds elapsed. After you create the kthread, bind it to a specific CPU (taken in from command-line) using `kthread_bind` and then wake it up using `wake_up_process`. Run `htop` to view the CPU getting used up.
- ```
[can do] (module) {process/kthread_cpu.c}
```
- 13** Write a module to swap two registered signal handlers. First write a program in C to register two signal handlers, one for handling SIGUSR1 and the other for SIGUSR2. Then load the module which simply swaps the two handlers, given the process's pid as command-line argument. Send the respective signals using `kill` to check your module's correctness. (HINT: you might want to dig deeper into `sighand` member of `task_struct`).
- ```
[can do] (module) {process/sighand.c,sighand_helper.c}
```
- 14** Write a module to make a process immune from all signals, including SIGKILL and SIGSTOP. That is, the process should continue execution even when these signals are delivered to it! (HINT: the hack lies in setting one of the process flags! the process has to lie about its status!). When you have done the needful, how would you now kill the process, without rebooting the system?
- ```
[can do] (module) {process/block_kill.c,block_kill_helper.c}
```
- 15** Write a module which locks the `task_struct` of a process using `alloc_lock` when loaded, and unlocks it when unloaded. The process id must be passed as a command-line argument. After loading the module, kill the process. Why did it not die, and why has it halted? Unload the module after a few seconds and see what happens. Then, check the dmesg log buffer to see the soft lockup message. Using the call trace in the debug output, find the reason for this. (HINT: this has something to do with another kernel function sleeping on `alloc_lock` that your module had held earlier). (HINT: kernel provides a function in `sched.h` for the required locking)
- ```
[can do] (module) {process/lockit.c,lockit_helper.c}
```
- 16** Use the module written in the exercise above to capture the exit information of a process. That is, print the `exit_code` and `exit_signal` fields of the process descriptor after the process has been signalled a kill but before it exits. Write a C program which prints something periodically, to test the module. Deliver SIGINT and SIGKILL to it and see the exit code and signal. Then, have the program register a signal handler for SIGINT which simply calls `exit` system call with an arbitrary number such as 2 or 5. Load the module again and see the difference in the value of `exit_code`. Why does it show such a value? Inspect the definition of `exit` system call in the

kernel source to find out the reason.

```
[can do] (module) {process/exit.c,exit_helper.c}
```

- 17** Write a program in C to spawn a few pthreads. Give a name to each of the pthread and make them all wait for user input. Then, write a kernel module which takes a pid as the command-line argument and prints the following information about each of its threads: thread/process id, name, pointer to `mm`, pointer to `stack` and pointer to `signal`. Reason about the values of `mm` and `stack` across all the threads. (HINT: you may want to use the `for_each_thread` macro for iterating through all the threads).

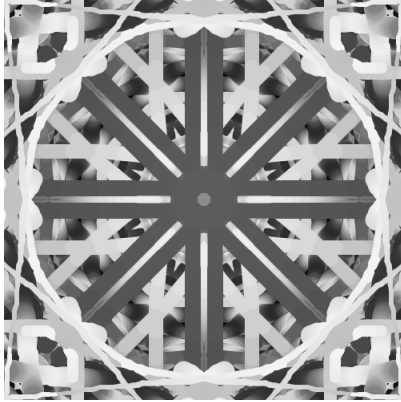
```
[can do] (module) {process/threads.c,threads_helper.c}
```

- 18** Write a program in C to fork a child process and then make it wait for user input. Then write a module to print the process group ID and session group ID of the child process created. Also print its group leader's pid. Reason about the output.

```
[can do] (module) {process/pids.c,pids_helper.c}
```

- 19** Write a module to print the names of all the open files associated with a process whose pid is passed as a command-line argument. Verify by looking at the output of `ls -lsof -p <pid>`. Explain why the first three file names output by your module are numbers.

```
[can do] (module) {process/files.c,files_helper.c}
```



7

Memory

Before delving into kernel memory allocation, let us go over the structure of the process virtual address space (VAS). A process' VAS is split between the user and kernel. The size of the individual VASes is specified using the `User:Kernel :: u:k` ratio, which is called the VM split. The virtual address where the kernel VAS (also called the kernel segment) begins is given by the `PAGE_OFFSET` configuration. The VM split can also be modified during the kernel build process, using configuration symbols like `CONFIG_VMSPLIT_3G`, `CONFIG_VMSPLIT_2G`, `CONFIG_VMSPLIT_1G` (any one of these will be set). For example, the Intel x86-32 process has a 3 : 1 VM split.

The VM split on 64-bit systems is not that straightforward because all the 64 bits are not used for addressing (64 bits correspond to a VAS of 16 EB of RAM!). For the x86_64 system, we only use the 48 least significant bits for addressing. Do not be under the impression that the upper 16 bits are useless! They are treated as a sign-extension, but also help you determine which segment a virtual address corresponds to: the MSB bits are set to 1 for the kernel VAS and 0 for the user VAS.

Note: It should be mentioned that these virtual addresses are not offsets from 0, but bitmasks that are used by the memory management unit (MMU) which with access to the kernel page tables, perform address translation.

To conclude, a process' VAS consists of a user-mode and kernel segment. While every process has a unique user-mode VAS, the kernel segment is shared. The size of user-mode and kernel segments are decided by the VM split. The `PAGE_OFFSET` macro points to the base of the kernel segment.

How does one examine the user VAS? The `procfs` exposes a process' user VAS through the `/proc/<PID>/maps` pseudo-file. As an example, let us try view the segment mapping of the process `self`,

```
$ cat /proc/self/maps
5648d9541000-5648d9543000 r--p 00000000 103:07 18612376 /usr/bin/cat
5648d9543000-5648d9548000 r-xp 00002000 103:07 18612376 /usr/bin/cat
5648d9548000-5648d954b000 r--p 00007000 103:07 18612376 /usr/bin/cat
5648d954b000-5648d954c000 r--p 00009000 103:07 18612376 /usr/bin/cat
5648d954c000-5648d954d000 rw-p 0000a000 103:07 18612376 /usr/bin/cat
5648db480000-5648db4a1000 rw-p 00000000 00:00 0 [heap]
7fac05c03000-7fac05c25000 rw-p 00000000 00:00 0
7fac05c25000-7fac06a03000 r--p 00000000 103:07 18617441 /usr/lib/locale/
locale-archive
7fac06a03000-7fac06a25000 r--p 00000000 103:07 18618340 /usr/lib/x86_64-
linux-gnu/libc-2.31.so
7fac06a25000-7fac06b9d000 r-xp 00022000 103:07 18618340 /usr/lib/x86_64-
linux-gnu/libc-2.31.so
7fac06b9d000-7fac06beb000 r--p 0019a000 103:07 18618340 /usr/lib/x86_64-
linux-gnu/libc-2.31.so
7fac06beb000-7fac06bef000 r--p 001e7000 103:07 18618340 /usr/lib/x86_64-
```

```

linux-gnu/libc-2.31.so
7fac06bef000-7fac06bf1000 rw-p 001eb000 103:07 18618340 /usr/lib/x86_64-
linux-gnu/libc-2.31.so
7fac06bf1000-7fac06bf7000 rw-p 00000000 00:00 0
7fac06c17000-7fac06c18000 r-p 00000000 103:07 18618120 /usr/lib/x86_64-
linux-gnu/ld-2.31.so
7fac06c18000-7fac06c3b000 r-xp 00001000 103:07 18618120 /usr/lib/x86_64-
linux-gnu/ld-2.31.so
7fac06c3b000-7fac06c43000 r-p 00024000 103:07 18618120 /usr/lib/x86_64-
linux-gnu/ld-2.31.so
7fac06c44000-7fac06c45000 r-p 0002c000 103:07 18618120 /usr/lib/x86_64-
linux-gnu/ld-2.31.so
7fac06c45000-7fac06c46000 rw-p 0002d000 103:07 18618120 /usr/lib/x86_64-
linux-gnu/ld-2.31.so
7fac06c46000-7fac06c47000 rw-p 00000000 00:00 0
7ffe963d1000-7ffe963f2000 rw-p 00000000 00:00 0 [stack]
7ffe963f4000-7ffe963f8000 r-p 00000000 00:00 0 [vvar]
7ffe963f8000-7ffe963fa000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 -xp 00000000 00:00 0 [vsyscall]

```

The keyword `self` represents the calling process (the process that will be run), `cat` in this case.

Every line in the output represents a segment or mapping in the VAS. Each line can be interpreted as follows: the first field is the range of user virtual addresses (UVAs) used by the segment in the form `start_uva-end_uva`. The second field is the mode (in `rwX` format) and mapping of the segment. Mapping indicates if the segment is private to a user process or shared. This is set up by the `flags` parameter to the `mmap` system call. The next field indicates the offset from the beginning of the file that is being mapped to the segment, `start-off`. This is only valid for file mappings and not anonymous ones. Anonymous mappings like the one for the stack and heap will have this field set to all zeros. After `start-off`, we have the major number, minor number field, `mj:mn`. The major number is used by the kernel to identify the driver associated with the device. It is possible for a driver to be responsible for multiple devices and hence, the minor number provides a way for the driver to differentiate between different devices it controls. Just like `start-off`, this field is only valid for file mappings and will be `00:00` for anonymous mappings. The penultimate field refers to the inode number of the image file that is mapped into the process VAS. Again, this is only valid for file mappings and will be `0` otherwise. In fact, this is one of the ways to determine if a mapping is file-based or anonymous! The last field is the path of the file being mapped.

The kernel abstracts these segments/mappings using a metadata structure, called the **Virtual Memory Area** (VMA). Only user VAS segments have VMAs! Kernel segments do not have a corresponding VMA. The VMAs of a process are organised as a red-black tree and can be accessed using `current->mm->mmap`. We have reproduced some portions of VMA for convenience,

```

struct vm_area_struct {
    unsigned long vm_start;
    unsigned long vm_end;

    struct vm_area_struct *vm_next, *vm_prev;

    struct rb_node vm_rb;

    unsigned long rb_subtree_gap;

    struct mm_struct *vm_mm;
    pgprot_t vm_page_prot;
    unsigned long vm_flags;

```



```

struct {
    struct rb_node rb;
    unsigned long rb_subtree_last;
} shared;

struct list_head anon_vma_chain;
struct anon_vma *anon_vma;

const struct vm_operations_struct *vm_ops;

unsigned long vm_pgoff;
struct file * vm_file;
void * vm_private_data;

struct vm_region *vm_region;
...
};

```

We now take a look at the kernel VAS of a process. Even though the kernel segment memory layout is highly architecture-specific, there are some common features. The kernel segment can be divided into the following regions,

- The lowmem region: The lowmem region is immediately after the user VAS, starting from address `PAGE_OFFSET` and is where your computer's RAM get mapped into the kernel. Each physical page frame gets mapped into a kernel page on a one-to-one basis. Therefore, the size of the lowmem region equals the amount of physical RAM available on your system. These virtual addresses are also sometimes referred to as kernel logical addresses. This region has the property that any logical address is at a fixed offset from the physical address in RAM, the offset being given by the `PAGE_OFFSET` macro. The kernel provides the `virt_to_phys()` and `phys_to_virt()` APIs to perform virtual-to-physical and physical-to-virtual address translation. These can only be used in the kernel lowmem region although it is recommended that you avoid using these methods.
- The `vmalloc` region: This region of the kernel VAS is completely virtual and device drivers and the kernel can allocate contiguous (virtual) memory from this region using `vmalloc`. Note that the memory allocated is virtually contiguous, but not physically contiguous, unlike the lowmem region. The macros `VMALLOC_START` and `VMALLOC_END` denote the starting and ending virtual address of this region.
- The kernel modules space: This is where the static text and data segments of modules are allocated. Therefore, whenever you perform `insmod`, the `init_module` system call allocates memory from this region of the kernel segment using `vmalloc`. The macros, `MODULES_VADDR` and `MODULES_END` denote the starting and ending virtual address of this region.

Address Space Layout Randomization (ASLR)

As described above, suppose you run `cat /proc/self/maps` to view the user VAS of the `self` process. What if you run the command again? It turns out that the addresses that you see would have changed! (remember, these are virtual addresses, not actual addresses) This is because of a kernel feature called **Address Space Layout Randomization**. ASLR was introduced for security purposes and when enabled (it is enabled by default), the starting addresses of the stack, heap, shared libraries, `mmap`-based allocations are randomized when the process is initiated. When one talks about ASLR, they are generally referring

to usermode ASLR. The kernel provides the `/proc/sys/kernel/randomize_va_space` within `procfs` that allows you to enable/disable ASLR. You can also disable ASLR at boot time, by passing the `norandmaps` parameter to the kernel (via `GRUB`).

Just like usermode ASLR, we can also randomize the layout of the kernel segment... to a certain extent. in KASLR (Kernel ASLR), the starting addresses of the kernel and module code will be randomized by a page-aligned offset from the base of RAM. The `CONFIG_RANDOMIZE_MEMORY` configuration lets you enable/disable KASLR. For an x86 system, kernel documentation in `Documentation/x86/x86_64/mm.txt` states that enabling the configuration will randomize the direct mapping of physical memory, `vmalloc/ioremap` and virtual memory maps. However, their order in the kernel VAS is preserved. Just like usermode ASLR, you can disable KASLR at runtime using the `nokaslr` parameter (the `kaslr` parameter turns on KASLR).

RAM Organisation

Let us now go over how the kernel views RAM. RAM is divided into nodes, nodes are divided into zones which in turn consist of page frames. Nodes abstract the concept of a physical bank of RAM, which are associated with one or more cores (nodes are abstracted in the kernel as the `pglist_data` structure which is referenced using its type definition `pg_data_t` in `include/linux/mmzone.h`). Each processor is connected to a RAM controller. The RAM controllers are connected to physical banks of RAM and are also interconnected which lets a core access other banks. It is evident that we can get a boost in performance by allocating memory on the RAM which is nearest to the core (which is running the thread). This is called a NUMA model, which stands for non-uniform memory access. To be able to talk about NUMA, the system should have multiple cores and multiple physical banks of RAM (if there was only one bank, there is no concept of the kernel preferentially allocating memory!). However, Linux treats all systems as NUMA, even those that only have a single bank of RAM, which are referred to as fake-NUMA systems. You can observe this using the `numactl` utility by running `numactl -hardware`. Running this on my PC gives,

```
$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 15752 MB
node 0 free: 11931 MB
node distances:
node    0
 0:    10
```

which confirms the presence of a single node, with all 8 cores (including hyperthreading) being associated to that bank of RAM.

Zones represent different ranges within memory. These are abstracted in the kernel using `zone_struct` (which is referenced as `zone`). Zones in turn are divided into page frames. The number and name of zones is determined by the kernel at boot time. We can list the zones by viewing the `buddyinfo` file in `procfs`. For example, examining the contents on an x86 computer with 16GB RAM gives,

```
$ cat /proc/buddyinfo
Node 0, zone    DMA      0      0      0      0      0      0      0      0      1
2      2
Node 0, zone    DMA32    9      6      9      6      5      3      6      5      4
6      232
Node 0, zone    Normal 10753 14638 7888 3929 1602 549 159 40 19
6      2494
```

This also confirms the presence of a single node with 3 zones, `DMA`, `DMA32`, and `Normal`.

Memory Allocators

To handle the allocation/deallocation of memory, the kernel uses a Page Allocator (PA), also known as a Buddy System Allocator (BSA). However, we will later see that this is not the only way to manage memory and the kernel also implements a Slab Allocator to alleviate some of the issues of the BSA. The slab allocator used in the kernel is called SLUB (it uses an unqueued slab allocator-based mechanism) and is built on top of the PA interface. Therefore, the entire Linux kernel (except the memory management unit itself) uses the page allocator for memory allocation/deallocation. Note that the page allocator gets its memory from the `lowmem` region of the kernel segment.

Page Allocator

Before learning about the PA interface, let us go over the Buddy System Algorithm (hence the name, Buddy System Allocator), which is used to implement the page allocator. To implement the Buddy System Algorithm, we must first come up with an efficient way to store the list of free page frames in RAM. This is accomplished using a data structure called the freelist. The freelist is an array of circular, doubly linked lists. Each list in the array has a corresponding order, such that raising two the order gives you the number of pages in a single entry of the corresponding linked list. The order varies from 0 to `MAX_ORDER - 1`, where the `MAX_ORDER` macro is architecture specific. Each entry in the circular doubly linked list points to a set of free contiguous page frames in RAM, where the size of the chunk is as given using the order of the list. The linked list entries are not actual physical memory frames, but rather their metadata structures in the form of `struct page`. Recall we had printed the contents of `/proc/buddyinfo` and had seen a bunch of numbers on each line. These numbers represent the size of each linked list in the freelist data structure. By counting the number of numbers on a line, you can determine `MAX_ORDER`! For the output above, we can verify that `MAX_ORDER` equals 11, which is typical for x86 and ARM systems. Finally, the kernel maintains separate freelists for each node:zone pair.

Next, let us understand the working of the Buddy System Algorithm with a simple example. We assume that the size of each page is 4kB. Suppose a kernel subsystem or device driver requests the page allocator for 64kB of memory. To fulfill this request, the page allocator will do the following:

1. For 64kB of memory, the allocator needs to find a set of 16 contiguous page frames. This will be present in the linked list with order 4. So, the allocator will scan through the corresponding list in the freelist of the current node:zone (remember that there are separate freelists for each node:zone), searching for some free entry.
2. If it finds such an entry, the allocator allocates that chunk of memory to the requesting party and removes it from the freelist.
3. In case there is no entry in the list with order 4, the allocator searches through the order 5 list. If the list is not empty, the allocator removes one of the chunks from the list for use. Since this chunk has a size greater than what was requested, the chunk is split in half, a part of which goes to the requesting party while the rest is added to the order 4 list.

4. In case there is no entry in the list with order 5, the allocator continues the process with the order 6 list and so on. In case all lists are empty, the request fails.

We are able to slice memory in this way only because the freelist guarantees that each chunk of memory is physically contiguous! Why is this called the Buddy System Algorithm? Whenever you cut a chunk of memory and allocate one of the resulting chunks, the other free chunk is called a buddy block. Since we typically deal with chunk sizes that are powers of two, this is also called the binary buddy system. When a block of memory is to be freed, it looks for the possibility of merging with its neighbouring blocks. If they are free, the allocator merges the two blocks, continuing this process until no more merging is possible. This prevents external fragmentation. However, the buddy system algorithm suffers from internal fragmentation within a chunk of memory. For example, if you request for 49kB of memory, the allocator will return a chunk of size 64kB, which means that 15kB of memory will be unused! There are methods to mitigate these effects via the `alloc_pages_exact()` and `free_pages_exact()` APIs. The implementation of this algorithm can be found in `mm/page_alloc.c:__alloc_pages_nodemask()`.

We had mentioned that memory zones (represented by `zone_t`) are divided into pages. Looking into the `zone` data structure, we observe that the kernel maintains a set of free areas in memory using an array of `struct free_area`. This is consistent with our statement that there is a separate freelist for each node:zone. `free_area` implements the circular doubly linked list and maintains the number of free page frames. One more thing, you would have noticed that the `free_area` structure maintains multiple freelists, corresponding to different page migration types. This was introduced in the 2.6.24 kernel to handle complications arising to prevent fragmentation. You can try printing the contents of `/proc/pagetypeinfo` to see the different page migration types along with the state of freelists.

Page Allocator APIs

All the routines described below can be found in `include/linux/gfp.h`.

- The `__get_free_page()` method allocates one free page and is just a wrapper around the `__get_free_pages()` API. The memory locations are initialised randomly. It returns a pointer to the allocated memory's kernel virtual address.
- The `__get_free_pages()` method allocates 2^{order} physically contiguous page frames. The memory locations are initialised to random values. It returns a pointer to the allocated memory's kernel virtual address.
- The `get_zeroed_page()` method is just like the methods above, exact it also initialises all memory locations to ASCII `0`. It returns a pointer to the allocated memory's kernel virtual address.
- The `alloc_page()` method allocates exactly one page frame. The memory allocated will have random initial values. Unlike the previous methods, this returns a pointer to the allocated memory's `page` metadata structure, which can be converted to a kernel virtual address using the `page_address()` API.
- The `alloc_pages()` method allocates 2^{order} physically contiguous page frames. The memory allocated will have random initial values. It returns a pointer to the allocated memory's `page` metadata structure (just like `alloc_page()`), which can be converted to a kernel virtual address using `page_address()`.

All the routines accept 2 parameters: GFP flags or bitmask, and the order of the freelist from which page frames are to be allocated. Note that when you use the `alloc_page()` or `alloc_pages()` API, the function returns a pointer to the `page` data structure corresponding to an actual page frame in RAM (the kernel

keeps track of every single page frame in RAM!). To access the actual memory chunk just allocated, you can use the `page_address()` API on the `page` structure returned.

```

struct page {
    unsigned long flags;

    union {
        struct address_space *mapping;
        void *s_mem;
        atomic_t compound_mapcount;
    };

    union {
        pgoff_t index;
        void *freelist;
    };

    union {
        _slub_counter_t counters;
        unsigned int active;
        struct {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };
        int units;

        struct {
            atomic_t _mapcount;
            atomic_t _refcount;
        };
    };

    union {
        struct list_head lru;
        struct dev_pagemap *pgmap;
        struct {
            struct page *next;
            int pages;
            int pobjects;
        };

        struct rcu_head rcu_head;

        struct {
            unsigned long compound_head;
            unsigned char compound_dtor;
            unsigned char compound_order;
        };

        struct {
            unsigned long __pad;
            pgtable_t pmd_huge_pte;
        };
    };

    union {
        unsigned long private;
        spinlock_t *ptl;
    };
};

```

```

    struct kmem_cache *slab_cache;
};

void *virtual;
...
};

```

While allocating memory, you will typically either use the `GFP_KERNEL` or `GFP_ATOMIC` flag as the GFP bitmask. The idea is that you are supposed to use `GFP_KERNEL` in process context, where it is safe to sleep, while you should use `GFP_ATOMIC` in interrupt contexts or when sleeping is not safe. By sleeping, we mean that a process/thread is in a sleep state where it is waiting for the occurrence of some event that will cause it to become active. We had seen that you can determine the kind of context using `in_task()`, `in_atomic()` macros and you must be mindful of this while allocating memory. Besides `GFP_KERNEL` and `GFP_ATOMIC`, there are other flags which can be found in `include/linux/gfp.h`.

Complementary to allocating pages, is freeing pages and the kernel provides several APIs for the same,

- The `free_page()` method frees exactly one page and is a wrapper around the `free_pages()` API.
- The `free_pages()` method allows you to free multiple pages and is a wrapper around the `__free_pages()` API.
- The `__free_pages()` method is the one that does all the work, and unlike the methods above, it accepts a pointer to the `page` metadata structure of the to-be-freed memory.

The preceding APIs are really just wrappers around `__free_pages()`, which is defined in `mm/page_alloc.c`. While `free_pages()` accepts the starting address of the memory chunk to be freed, `__free_pages()` (which actually does the work) accepts a pointer to the page structure of chunk which is to be freed.

In general, freeing memory is a tricky affair which is prone to several errors including memory leakage, double-free and use-after-free (UAF). A few things you can keep in mind to avoid such problems are,

-

The major drawback of using the page allocator APIs we described till now is high memory wastage due to internal fragmentation. Thankfully, there is an alternative method which leads to far less waste when allocating large chunks of memory. This involves using the `alloc_pages_exact()` and `free_pages_exact()` APIs. `alloc_pages_exact()` accepts as an argument, the size of memory requested and the returned chunk is guaranteed to be physically contiguous. How does this reduce internal fragmentation? The function begins by allocating a memory chunk according to the buddy system algorithm, however, it frees a part of the entire chunk so that the effective memory allocated equals the size of memory requested. For example, if you request for 28 kB of memory, `alloc_pages_exact()` will first allocate a 32 kB chunk and then free memory from 29 kB to 32 kB. To free memory allocated, you need to use the complementary `free_pages_exact()` API, which accepts a pointer to the beginning of the memory chunk to be freed along with the GFP flag.

Slab Allocator

The slab allocator or slab cache is built upon the page allocator and serves two primary purposes: object caching and mitigating internal fragmentation by the page allocator.

Caching is a very popular technique to achieve faster performance. In the kernel, there are several data structures that are allocated and deallocated frequently including the networks stack's socket buffer `sk_buff`

and a process' `task_struct`. Therefore, these data structures are pre-allocated into several slab caches at boot time. This only happens once (at boot time) and ensures that unlike traditional heap-based allocators, memory remains relatively unfragmented. You can examine the current state of slab caches by printing the contents of `/proc/slabinfo`. (Should I add an output example?)

You can also view the amount of memory being used by the slab cache by searching for Slab in `/proc/meminfo`,

```
$ grep "Slab" /proc/meminfo
Slab:                266908 kB
```

The slab cache might use significant portions of memory to improve performance, however this usage is intelligently reduced in cases of high memory pressure. The page cache is one such example that utilises a large amount of RAM.

One of the main reasons for internal fragmentation by the page allocator was the granularity at which requests are handled - memory is always allocated at the page level! The slab cache attempts to alleviate this problem by servicing requests using fragments of pages.

Slab Allocator APIs

At its core, all slab allocations/deallocations use two functions, while the rest are defined for the sake of convenience. These are the `kmalloc` and `kzalloc` functions, which are defined in `include/linux/slab.h`. Both functions accept two arguments: the size of memory requested and GFP flags. The API will return a pointer to the start of the memory chunk (also called a slab) just allocated (this is a kernel logical address). `kmalloc` and `kzalloc` are guaranteed to return a physically contiguous memory chunk. Another benefit that they offer is that the return address is guaranteed to be cacheline-aligned for performance. Transfers between the CPU and memory occur at the granularity of cache lines. The size of cache lines varies with architecture and can be inspected using the `getconf` utility. Typical cache line sizes are 64 bytes.

What is the difference between `kmalloc` and `kzalloc`? On calling `kmalloc`, the contents of the slab returned are random, which is not recommended. On the other hand, `kzalloc` ensures all contents are initialised to zero, which makes it the method of choice (even at the cost of performance taking a slight hit).

The counterpart to `kmalloc` and `kzalloc` for freeing memory is `kfree`. `kfree` accepts the pointer returned by `kmalloc/kzalloc` that you are looking to free. Note that `kfree` simply returns the slab back to the corresponding cache. Whatever was written to the slab remains! For security reasons, it is recommended that you overwrite the contents of the slab using the `kzfree` API.

We know that `kmalloc/kzalloc` get their memory from the slab caches. After printing the contents of `/proc/slabinfo`, we know that there are several slab caches for frequently used data structures. So exactly which slab caches are the APIs getting their memory from? Running the `vmstat` utility lends an answer to the question,

```
$ sudo vmstat -m | grep "kmalloc"
...
kmalloc-8k          299    304   8192     4
kmalloc-4k         2401   2448   4096     8
kmalloc-2k         1869   1920   2048    16
kmalloc-1k         2864   3040   1024    32
kmalloc-512       13669  13792    512    32
kmalloc-256       6914   7520    256    32
```

<code>kmalloc-192</code>	7670	7749	192	21
<code>kmalloc-128</code>	3136	3136	128	32
<code>kmalloc-96</code>	4284	4284	96	42
<code>kmalloc-64</code>	24647	26816	64	64
<code>kmalloc-32</code>	31986	34048	32	128
<code>kmalloc-16</code>	25344	25344	16	256
<code>kmalloc-8</code>	11776	11776	8	512

where the output has been truncated for convenience (you could have also searched for these slab caches in `/proc/slabinfo`). As you can see, there are several slab caches which are responsible for servicing requests between 8 bytes to 8 kB. The range of sizes available are also architecture specific (the output above is for an x86 system with 16 GB RAM). There are also other cache types like `dma-kmalloc-N`, `kmalloc-rcl-N` and `kmalloc-cg-N`.

Tip: It is considered a good practice to keep all context information associated with a device driver in a metadata structure.

Should I add a section on size of maximum request that can be served by the slab and page allocators?
Should I add a section on resource-managed memory allocation APIs for device drivers?

There are at least three different implementations of a slab allocator in the kernel and the one that is used depends on the value of configuration symbols: `CONFIG_SLAB`, `CONFIG_SLUB`, `CONFIG_SLOB`.

Structs and Flags


```
struct mm_struct {
    struct vm_area_struct *mmap;
    struct rb_root mm_rb;
    u32 vmacache_seqnum;

    unsigned long mmap_base;
    unsigned long mmap_legacy_base;

    unsigned long task_size;
    unsigned long highest_vm_end;
    pgd_t * pgd;

    atomic_t mm_users, mm_count;

    atomic_long_t pgtables_bytes;

    int map_count;

    spinlock_t page_table_lock;
    struct rw_semaphore mmap_sem;

    struct list_head mmlist;

    unsigned long hiwater_rss, hiwater_vm;

    unsigned long total_vm, locked_vm, pinned_vm;
    unsigned long data_vm, exec_vm, stack_vm;
    unsigned long def_flags;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

    struct mm_rss_stat rss_stat;

    mm_context_t context;

    unsigned long flags;

    struct core_state *core_state;

    struct file __rcu *exe_file;

    pgtable_t pmd_huge_pte;

    atomic_t tlb_flush_pending;

    atomic_long_t hugetlb_usage;
    ...
};
```

Functions and Macros

Assignments

- 1 Write a module to print the following information about the kernel internals:

- Number of page table levels
- Page size
- Huge page size

```
[must do] (module) {memory/mem_values.c}
```

- 2 Write a module to count the number of processes and kernel threads in the system. (HINT: use the fact that kernel threads will have no personal memory image allocated and refer to the `mm` field of `task_struct`).

```
[must do] (module) {memory/numproc.c}
```

- 3 Write a module to calculate the total virtual set size (VSS) of a process whose pid is passed as a command-line argument. Also, count the total number of virtual memory areas (VMAs) the process has. Verify your output by visiting the `/proc` entries for the pid and do the following. Get the number of VMAs from the output of `maps` file. Get the VSS from either the output of `status` file, or from the output of `top/htop`. You will see that your VMA count will be one less than that of `/proc/<pid>/maps` file due to `vsyscall`.

```
[must do] (module) {memory/vma.c}
```

- 4 Write a basic program in C to `fork()` a child process. Have the child process run an infinite empty loop and make the parent `wait()` on the child. Then write a kernel module to print the information about the VMAs of the parent as well as the child. The module should take the pids of parent and child processes as command-line arguments. You must get the "same" output for parent and child. Why is that so?

Then modify the C program you wrote above to `exec()` the `sleep` command for very long time (say 1000 seconds) instead of an infinite loop. Load the module again and reason about the output you see now.

```
[must do] (module) {memory/pc.c}
```

- 5 Write a module to find out the resident set size (RSS) of a process whose pid is passed as a command-line argument to the module. The RSS calculated by the module should match with that of the output of `/proc/<pid>/smaps`. (HINT: you might need to skip a VMA with `pf` flag!).

```
[must do] (module) {memory/rss.c,mem_helper.c}
```

- 6 Write a module to change the contents of a page in memory. First, write a C program to create two memory mappings using `mmap` of 100 and 200 bytes, respectively. `memset` the two memory regions to all 'a's and all 'b's, respectively. Then, wait for some user input (simple `getchar` will work) and print the contents of the two memory regions. While the C program is waiting for user input, write the module which goes through all the pages of that process, finds those two pages which have all 'a's and all 'b's, respectively, and makes some changes. Load the module passing the pid as a command-line argument. Finally, provide some input to the C program and it should print the modified page

contents.

```
[must do] (module) {memory/mmap.c}
```

- 7 Write a C program to allocate/free some memory in an infinite loop. Then, write a module to write-protect all the VMAs of a process. Run the C program and pass its pid to the module. The process should receive a segmentation fault when it tries to access one of the VMAs.

```
[must do] (module) {memory/lock.c,lock_helper.c}
```

- 8 Write a module to implement a buddy allocation system. Your module should first acquire some pages, say 4, which will act as the main memory for your system. Then the module should read an array of integers from the command-line which will be the memory requests to this system. For each request, you must use best-fit allocation strategy. The smallest allowable block size should be 64 Bytes. Also calculate the amount of memory wasted here, i.e. internal fragmentation. You will need to store some metadata about your buddy system in a tree-like structure where each node represents a block. The nodes can have information like block size, data length, link to left block, link to right block etc. If the memory requests were 8000,3000,2000,500,500, then the output should be something like this:

```
[ +0.000017] Split (16384)
[ +0.000001] 8000 (8192)
[ +0.000001] Split (8192)
[ +0.000000] 3000 (4096)
[ +0.000001] Split (4096)
[ +0.000000] 2000 (2048)
[ +0.000000] Split (2048)
[ +0.000001] Split & full (1024)
[ +0.000001] 0 (1024)
[ +0.000000] 500 (512)
[ +0.000001] 500 (512)
[ +0.000000] Internal fragmentation: 1360 bytes out of 16384 bytes.
```

If the memory requests were 500,1000,2000,4000,8000,500,500, the last memory request should trigger an error.

```
[try to do] (module) {memory/buddy.c}
```

Suggested Reading

menuconfig / Kconfig

FLAGS

processes

```

/*
 * thread information flags
 * — these are process state flags that various assembly files
 * may need to access
 */
#define TIF_SYSCALL_TRACE      0 /* syscall trace active */
#define TIF_NOTIFY_RESUME     1 /* callback before returning to user */
#define TIF_SIGPENDING        2 /* signal pending */
#define TIF_NEED_RESCHED      3 /* rescheduling necessary */
#define TIF_SINGLESTEP        4 /* reenable singlestep on user return */
#define TIF_SSB                5 /* Reduced data speculation */
#define TIF_SYSCALL_EMU       6 /* syscall emulation active */
#define TIF_SYSCALL_AUDIT     7 /* syscall auditing active */
#define TIF_SECCOMP            8 /* secure computing */
#define TIF_USER_RETURN_NOTIFY 11 /* notify kernel of userspace return */
#define TIF_UPROBE             12 /* breakpointed or singlestepping */
#define TIF_PATCH_PENDING     13 /* pending live patching update */
#define TIF_NOCPUID            15 /* CPUID is not accessible in userland */
#define TIF_NOTSC              16 /* TSC is not accessible in userland */
#define TIF_IA32               17 /* IA32 compatibility process */
#define TIF_NOHZ               19 /* in adaptive nohz mode */
#define TIF_MEMDIE             20 /* is terminating due to OOM killer */
#define TIF_POLLING_NRFLAG     21 /* idle is polling for TIF_NEED_RESCHED */
#define TIF_IO_BITMAP          22 /* uses I/O bitmap */
#define TIF_FORCED_TF          24 /* true if TF in eflags artificially */
#define TIF_BLOCKSTEP          25 /* set when we want DEBUGCTLMSR_BTF */
#define TIF_LAZY_MMU_UPDATES   27 /* task is updating the mmu lazily */
#define TIF_SYSCALL_TRACEPOINT 28 /* syscall tracepoint instrumentation */
#define TIF_ADDR32             29 /* 32-bit address space on 64 bits */
#define TIF_X32                30 /* 32-bit native x86-64 binary */
#define TIF_FSCHECK            31 /* Check FS is USER_DS on return */

```

```

#define _PAGE_BIT_PRESENT      0 /* is present */
#define _PAGE_BIT_RW           1 /* writeable */
#define _PAGE_BIT_USER         2 /* userspace addressable */
#define _PAGE_BIT_PWT          3 /* page write through */
#define _PAGE_BIT_PCD          4 /* page cache disabled */
#define _PAGE_BIT_ACCESSED     5 /* was accessed (raised by CPU) */
#define _PAGE_BIT_DIRTY        6 /* was written to (raised by CPU) */
#define _PAGE_BIT_PSE          7 /* 4 MB (or 2MB) page */
#define _PAGE_BIT_PAT          7 /* on 4KB pages */
#define _PAGE_BIT_GLOBAL       8 /* Global TLB entry PPro+ */
#define _PAGE_BIT_SOFTW1       9 /* available for programmer */
#define _PAGE_BIT_SOFTW2      10 /* " */
#define _PAGE_BIT_SOFTW3      11 /* " */
#define _PAGE_BIT_PAT_LARGE    12 /* On 2MB or 1GB pages */
#define _PAGE_BIT_SOFTW4      58 /* available for programmer */
#define _PAGE_BIT_PKEY_BIT0    59 /* Protection Keys, bit 1/4 */
#define _PAGE_BIT_PKEY_BIT1    60 /* Protection Keys, bit 2/4 */
#define _PAGE_BIT_PKEY_BIT2    61 /* Protection Keys, bit 3/4 */

```

```
#define _PAGE_BIT_PKEY_BIT3 62 /* Protection Keys, bit 4/4 */  
#define _PAGE_BIT_NX      63 /* No execute: only valid after cpuid check */
```