**«AZƏRBAYCAN HAVA YOLLARI »**

**Qapalı Səhmdar Cəmiyyəti**

**MİLLİ AVİASİYA AKADEMİYASI**

**“Sistem proqramlaşdırması və əməliyyat sistemləri-2” fənni üzrə**

**KURS İŞİ**

**Mövzu: “**Driver development in Linux OS**”**

**Faкültə:** Aerokosmik

**Кafedra:** Kompüter sistemləri və proqramlaşdırma

**İxtisas:** Kompüter mühəndisliyi

**Qrup:** 1459i

**Tələbə:** Hümmətova Məryəm

**Müəllim:** Şahmalıyev Məmməd

**Kafedra müdiri:** Məlikov Ağası

**Qiymət:**

**Müdafiə tarixi: 31.05.22**

**BAКI – 2022**

**Driver development in Linux OS**

In [computing](https://en.wikipedia.org/wiki/Computing), a device driver is a [computer program](https://en.wikipedia.org/wiki/Computer_program) that operates or controls a particular type of [device](https://en.wikipedia.org/wiki/Peripheral) that is attached to a [computer](https://en.wikipedia.org/wiki/Computer) or [automaton](https://en.wikipedia.org/wiki/Automaton). A driver provides a software [interface](https://en.wikipedia.org/wiki/Interface_(computing)) to [hardware](https://en.wikipedia.org/wiki/Computer_hardware) devices, enabling [operating systems](https://en.wikipedia.org/wiki/Operating_system) and other computer programs to access hardware functions without needing to know precise details about the hardware being used.

A driver communicates with the device through the [computer bus](https://en.wikipedia.org/wiki/Computer_bus) or communications subsystem to which the hardware connects. When a [calling](https://en.wikipedia.org/wiki/System_call) program invokes a [routine](https://en.wikipedia.org/wiki/Subroutine) in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program.

Drivers are hardware dependent and operating-system-specific. They usually provide the [interrupt](https://en.wikipedia.org/wiki/Interrupt) handling required for any necessary asynchronous time-dependent hardware interface.

Writing a device driver requires an in-depth understanding of how the hardware and the software works for a given [platform](https://en.wikipedia.org/wiki/Computing_platform) function. Because drivers require low-level access to hardware functions in order to operate, drivers typically operate in a highly [privileged](https://en.wikipedia.org/wiki/Privilege_(computing)) environment and can cause system operational issues if something goes wrong. In contrast, most user-level software on modern [operating systems](https://en.wikipedia.org/wiki/Operating_system) can be stopped without greatly affecting the rest of the system. Even drivers executing in [user mode](https://en.wikipedia.org/wiki/User_mode) can crash a system if the device is [erroneously programmed](https://en.wikipedia.org/wiki/Erroneous_program). These factors make it more difficult and dangerous to diagnose problems.

The task of writing drivers thus usually falls to [software engineers](https://en.wikipedia.org/wiki/Software_engineer) or [computer engineers](https://en.wikipedia.org/wiki/Computer_engineer) who work for hardware-development companies. This is because they have better information than most outsiders about the design of their hardware. Moreover, it was traditionally considered in the hardware [manufacturer](https://en.wikipedia.org/wiki/Manufacturer)'s interest to guarantee that their clients can use their hardware in an optimum way. Typically, the [Logical Device Driver](https://en.wikipedia.org/w/index.php?title=Logical_Device_Driver&action=edit&redlink=1) is written by the operating system vendor, while the [Physical Device Driver](https://en.wikipedia.org/w/index.php?title=Physical_Device_Driver&action=edit&redlink=1) is implemented by the device vendor. However, in recent years, non-vendors have written numerous device drivers for proprietary devices, mainly for use with [free and open source](https://en.wikipedia.org/wiki/Free_and_open_source_software) [operating systems](https://en.wikipedia.org/wiki/Operating_system). In such cases, it is important that the hardware manufacturer provide information on how the device communicates. Although this information can instead be learned by [reverse engineering](https://en.wikipedia.org/wiki/Reverse_engineering), this is much more difficult with hardware than it is with software.

In [Linux](https://en.wikipedia.org/wiki/Linux_kernel) environments, programmers can build device drivers as parts of the [kernel](https://en.wikipedia.org/wiki/Linux_kernel), separately as loadable [modules](https://en.wikipedia.org/wiki/Loadable_kernel_module), or as user-mode drivers . [Makedev](https://en.wikipedia.org/wiki/Makedev) includes a list of the devices in Linux, including ttyS, lp, hd, loop, and sound .

The advantage of loadable device drivers is that they can be loaded only when necessary and then unloaded, thus saving kernel memory.

**Contents:**

Getting started with the Linux kernel module

Creating a kernel module

Registering a character device

The file\_operations structure

The printk function

Using memory allocated in user mode

Building the kernel module

Loading and using the module

Conclusion

Resources

**Getting started with the Linux kernel module**

The Linux kernel is written in the C and Assembler programming languages. C implements the main part of the kernel, while Assembler implements architecture-dependent parts. That’s why we can use only these two languages for Linux device driver development. We cannot use C++, which is used for the Microsoft Windows kernel, because some parts of the Linux kernel source code may include keywords from C++ , while in Assembler we may encounter lexemes such as ‘ : : ’.

There are two ways of programming a Linux device driver:

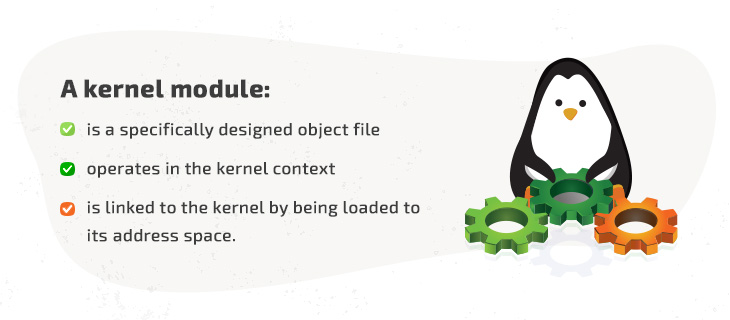
Compile the driver along with the kernel, which is monolithic in Linux.

Implement the driver as a kernel module, in which case you won’t need to recompile the kernel.

A module is a specifically designed object file. When working with modules, Linux links them to the kernel by loading them to the kernel address space.

Module code has to operate in the kernel context. This requires a developer to be very attentive. If a developer makes a mistake when implementing a user-level application, it will not cause problems outside the user application in most cases. But mistakes in the implementation of a kernel module will lead to system-level issues.

Luckily for us, the Linux kernel is resistant to non-critical errors in module code. When the kernel encounters such errors, it displays the oops message — an indicator of insignificant malfunctions during Linux operation. After that, the malfunctioning module is unloaded, allowing the kernel and other modules to work as usual. In addition, you can analyze logs that precisely describe non-critical errors.



The kernel and its modules represent a single program module and use a single global namespace. In order to minimize the namespace, you must control what’s exported by the module. Exported global characters must have unique names and be cut to the bare minimum. A commonly used workaround is to simply use the name of the module that’s exporting the characters as the prefix for a global character name.

**Creating a kernel module**

We’ll start by creating a simple prototype of a kernel module that can be loaded and unloaded. We can do that with the following code:

*#include <linux/init.h>*

*#include <linux/module.h>*

*static* *int* *my\_init(void)*

*{*

*return*  *0;*

*}*

*static* *void* *my\_exit(void)*

*{*

*return;*

*}*

*module\_init(my\_init);*

*module\_exit(my\_exit);*

The my\_init function is the driver initialization entry point and is called during system startup or when the module is inserted into the kernel. The my\_exit function is the driver exit point. It’s called when unloading a module from the Linux kernel. This function has no effect if the driver is statically compiled into the kernel.

These functions are declared in the linux/module.h header file. The my\_init and my\_exit functions must have identical signatures such as these:

*int* *init(void);*

*void* *exit(void);*

Now our simple module is complete. These operations will be useful for Linux kernel driver development.

**Registering a character device**

Device files are usually stored in the/dev folder. They facilitate interactions between the user space and the kernel code. To make the kernel receive anything, you can just write it to a device file to pass it to the module serving this file. Anything that’s read from a device file originates from the module serving it.

There are two groups of device files:

Character files — Non-buffered files that allow you to read and write data character by character.

Block files — Buffered files that allow you to read and write only whole blocks of data.

Linux systems have two ways of identifying device files:

Major device numbers identify modules serving device files or groups of devices.

Minor device numbers identify specific devices among a group of devices specified by a major device number.

We can define these numbers in the driver code, or they can be allocated dynamically. In case a number defined as a constant has already been used, the system will return an error. When a number is allocated dynamically, the function reserves that number to prevent other device files from using the same number.

To register a character device, we need to use the register\_chrdev function:

*int* *register\_chrdev (unsigned int*   *major,*

*const* *char* *\*   name,*

*const* *struct* *file\_operations \* fops);*

Here, we specify the name and the major number of a device to register it. After that, the device and the file\_operations structure will be linked. If we assign 0 to the major parameter, the function will allocate a major device number on its own. If the value returned is 0, this indicates success, while a negative number indicates an error. Both device numbers are specified in the 0–255 range.

The device name is a string value of the name parameter. This string can pass the name of a module if it registers a single device. We use this string to identify a device in the /sys/devices file. Device file operations such as read, write, and save are processed by the function pointers stored within the file\_operations structure. These functions are implemented by the module, and the pointer to the module structure identifying this module is also stored within the file\_operations structure

**The file\_operations structure**

In the Linux 5.3.0 kernel, the file\_operations structure looks like this:

*struct* *file\_operations {*

*struct* *module \*owner;*

*loff\_t (\*llseek) (struct* *file \*, loff\_t, int);*

*ssize\_t (\*read) (struct* *file \*, char* *\_\_user \*, size\_t, loff\_t \*);*

*ssize\_t (\*write) (struct* *file \*, const* *char* *\_\_user \*, size\_t, loff\_t \*);*

*ssize\_t (\*read\_iter) (struct* *kiocb \*, struct* *iov\_iter \*);*

*ssize\_t (\*write\_iter) (struct* *kiocb \*, struct* *iov\_iter \*);*

*int* *(\*iopoll)(struct* *kiocb \*kiocb, bool* *spin);*

*int* *(\*iterate) (struct* *file \*, struct* *dir\_context \*);*

*int* *(\*iterate\_shared) (struct* *file \*, struct* *dir\_context \*);*

*\_\_poll\_t (\*poll) (struct* *file \*, struct* *poll\_table\_struct \*);*

*long* *(\*unlocked\_ioctl) (struct* *file \*, unsigned int, unsigned long);*

*long* *(\*compat\_ioctl) (struct* *file \*, unsigned int, unsigned long);*

*int* *(\*mmap) (struct* *file \*, struct* *vm\_area\_struct \*);*

*unsigned long* *mmap\_supported\_flags;*

*int* *(\*open) (struct* *inode \*, struct* *file \*);*

*int* *(\*flush) (struct* *file \*, fl\_owner\_t id);*

*int* *(\*release) (struct* *inode \*, struct* *file \*);*

*int* *(\*fsync) (struct* *file \*, loff\_t, loff\_t, int* *datasync);*

*int* *(\*fasync) (int, struct* *file \*, int);*

*int* *(\*lock) (struct* *file \*, int, struct* *file\_lock \*);*

*ssize\_t (\*sendpage) (struct* *file \*, struct* *page \*, int, size\_t, loff\_t \*, int);*

*unsigned long* *(\*get\_unmapped\_area)(struct* *file \*, unsigned long, unsigned long, unsigned long, unsigned long);*

*int* *(\*check\_flags)(int);*

*int* *(\*flock) (struct* *file \*, int, struct* *file\_lock \*);*

*ssize\_t (\*splice\_write)(struct* *pipe\_inode\_info \*, struct* *file \*, loff\_t \*, size\_t, unsigned int);*

*ssize\_t (\*splice\_read)(struct* *file \*, loff\_t \*, struct* *pipe\_inode\_info \*, size\_t, unsigned int);*

*int* *(\*setlease)(struct* *file \*, long, struct* *file\_lock \*\*, void* *\*\*);*

*long* *(\*fallocate)(struct* *file \*file, int* *mode, loff\_t offset,*

*loff\_t len);*

*void* *(\*show\_fdinfo)(struct* *seq\_file \*m, struct* *file \*f);*

*#ifndef CONFIG\_MMU*

*unsigned (\*mmap\_capabilities)(struct* *file \*);*

*#endif*

*ssize\_t (\*copy\_file\_range)(struct* *file \*, loff\_t, struct* *file \*,*

*loff\_t, size\_t, unsigned int);*

*loff\_t (\*remap\_file\_range)(struct* *file \*file\_in, loff\_t pos\_in,*

*struct* *file \*file\_out, loff\_t pos\_out,*

*loff\_t len, unsigned int* *remap\_flags);*

*int* *(\*fadvise)(struct* *file \*, loff\_t, loff\_t, int);*

*} \_\_randomize\_layout;*

If this structure contains functions that aren’t required for your driver, you can still use the device file without implementing them. A pointer to an unimplemented function can simply be set to 0. After that, the system will take care of implementing the function and make it behave normally. In our case, we’ll just implement the read function.

As we’re going to ensure the operation of only a single type of device with our Linux driver, our file\_operations structure will be global and static. After it’s created, we’ll need to fill it statically like this:

*#include <linux/fs.h>*

*static* *struct* *file\_operations simple\_driver\_fops =*

*{*

*.owner   = THIS\_MODULE,*

*.read    = device\_file\_read,*

*};*

The declaration of the THIS\_MODULE macro is contained in the linux/export.h header file. We’ll transform the macro into a pointer to the module structure of the required module. Later, we’ll write the body of the function with a prototype, but for now we have only the device\_file\_read pointer to it:

ssize\_t device\_file\_read (struct file \*, char \*, size\_t, loff\_t \*);

The file\_operations structure allows us to develop several functions that will register and revoke the registration of the device file. To register a device file, we use the following code:

*static* *int* *device\_file\_major\_number = 0;*

*static* *const* *char* *device\_name[] = "Simple-driver";*

*int* *register\_device(void)*

*{*

*int* *result = 0;*

*printk( KERN\_NOTICE "Simple-driver: register\_device() is called.\n"* *);*

*result = register\_chrdev( 0, device\_name, &simple\_driver\_fops );*

*if( result < 0 )*

*{*

*printk( KERN\_WARNING "Simple-driver:  can\'t register character device with error code = %i\n", result );*

*return* *result;*

*}*

*device\_file\_major\_number = result;*

*printk( KERN\_NOTICE "Simple-driver: registered character device with major number = %i and minor numbers 0...255\n", device\_file\_major\_number );*

*return* *0;*

*}*

device\_file\_major\_number is a global variable that contains the major device number. When the lifetime of the driver expires, this global variable will be used to revoke the registration of the device file.

In the code above, we’ve added the printk function that logs kernel messages. Pay attention to the KERN\_NOTICE and KERN\_WARNING prefixes in all listed printk format strings. NOTICE and WARNING indicate the priority level of a message. Levels range from insignificant (KERN\_DEBUG) to critical (KERN\_EMERG), alerting about kernel instability. This is the only difference between the printk function and the printf library function.

**The printk function**

The printk function forms a string, which we add to the circular buffer. From there the klog daemon reads it and sends it to the system log. Implementing the printk allows us to call this function from any point in the kernel. Use this function carefully, as it may cause overflow of the circular buffer, meaning the oldest message will not be logged.

Our next step is writing a function for unregistering the device file. If a device file is successfully registered, the value of the device\_file\_major\_number will not be 0. This value allows us to revoke the registration of a file using the unregister\_chrdev function, which we declare in the linux/fs.h file. The major device number is the first parameter of this function, followed by a string containing the device name. The register\_chrdev and the unresister\_chrdev functions have similar contents.

To unregister a device, we use the following code:

*void* *unregister\_device(void)*

*{*

*printk( KERN\_NOTICE "Simple-driver: unregister\_device() is called\n"* *);*

*if(device\_file\_major\_number != 0)*

*{*

*unregister\_chrdev(device\_file\_major\_number, device\_name);*

*}*

*}*

The next step in implementing functions for our module is allocating and using memory in user mode. Let’s see how it’s done.

**Using memory allocated in user mode**

The read function we’re going to write will read characters from a device. The signature of this function must be appropriate for the function from the file\_operations structure:

ssize\_t (\*read) (struct file \*filep, char \*buffer, size\_t len, loff\_t \*offset);

Let’s look at the filep parameter — the pointer to the file structure. This file structure allows us to get necessary information about the file we’re working with, data related to this file, and more. The data we’ve read is allocated in the user space at the address specified by the second parameter — buffer. The number of bytes to be read is defined in the len parameter, and we start reading bytes from a certain offset defined in the offset parameter. After executing the function, the number of bytes that have been successfully read must be returned. Then we must refresh the offset.

To work with information from the device file, the user allocates a special buffer in the user-mode address space. Then, the read function copies the information to this buffer. The address to which a pointer from the user space points and the address in the kernel address space may have different values. That’s why we cannot simply dereference the pointer.

When working with these pointers, we have a set of specific macros and functions we declare in the linux/uaccess.h file. The most suitable function in our case is copy\_to\_user. Its name speaks for itself: it copies specific data from the kernel buffer to the buffer allocated in the user space. It also verifies if a pointer is valid and if the buffer size is large enough. Here’s the code for the copy\_to\_user prototype:

long copy\_to\_user( void \_\_user \*to, const void \* from, unsigned long n );

First of all, this function must receive three parameters:

A pointer to the buffer

A pointer to the data source

The number of bytes to be copied

If there are any errors in execution, the function will return a value other than 0. In case of successful execution, the value will be 0. The copy\_to\_user function contains the \_user macro that documents the process. Also, this function allows us to find out if the code uses pointers from the address space correctly. This is done using Sparse, an analyzer for static code. To be sure that it works correctly, always mark the user address space pointers as \_user.

Here’s the code for implementing the read function:

*#include <linux/uaccess.h>*

*static* *const* *char*    *g\_s\_Hello\_World\_string[] = "Hello world from kernel mode!\n\0";*

*static* *const* *ssize\_t g\_s\_Hello\_World\_size = sizeof(g\_s\_Hello\_World\_string);*

*static* *ssize\_t device\_file\_read(*

*struct* *file \*file\_ptr*

*, char* *\_\_user \*user\_buffer*

*, size\_t* *count*

*, loff\_t \*position)*

*{*

*printk( KERN\_NOTICE "Simple-driver: Device file is read at offset = %i, read bytes count = %u\n"*

*, (int)\*position*

*, (unsigned int)count );*

*/\* If position is behind the end of a file we have nothing to read \*/*

*if( \*position >= g\_s\_Hello\_World\_size )*

*return* *0;*

*/\* If a user tries to read more than we have, read only as many bytes as we have \*/*

*if( \*position + count > g\_s\_Hello\_World\_size )*

*count = g\_s\_Hello\_World\_size - \*position;*

*if( copy\_to\_user(user\_buffer, g\_s\_Hello\_World\_string + \*position, count) != 0 )*

*return* *-EFAULT;*

*/\* Move reading position \*/*

*\*position += count;*

*return* *count;*

*}*

With this function, the code for our driver is ready. Now it’s time to build the kernel module and see if it works as expected.

**Building the kernel module**

In modern kernel versions, the makefile does most of the building for a developer. It starts the kernel build system and provides the kernel with information about the components required to build the module.

A module built from a single source file requires a single string in the makefile. After creating this file, you only need to initiate the kernel build system with the obj-m := source\_file\_name.o command. As you can see, here we’ve assigned the source file name to the module — the \*.ko file.

If there are several source files, only two strings are required for the kernel build:

*obj-m := module\_name.o*

*module\_name-objs := source\_1.o source\_2.o … source\_n.o*

To initialize the kernel build system and build the module, we need to use the make –C KERNEL\_MODULE\_BUILD\_SYSTEM\_FOLDER M=`pwd` modules command. To clean up the build folder, we use the make –C KERNEL\_MODULES\_BUILD\_SYSTEM\_FOLDER M=`pwd` clean command.

The module build system is commonly located in /lib/modules/`uname -r`/build. Now it’s time to prepare the module build system. To build our first module, execute the make modules\_prepare command from the folder where the build system is located.

Finally, we’ll combine everything we’ve learned into one makefile:

*TARGET\_MODULE:=simple-module*

*# If we are running by kernel building system*

*ifneq ($(KERNELRELEASE),)*

*$(TARGET\_MODULE)-objs := main.o device\_file.o*

*obj-m := $(TARGET\_MODULE).o*

*# If we running without kernel build system*

*else*

*BUILDSYSTEM\_DIR:=/lib/modules/$(shell uname -r)/build*

*PWD:=$(shell pwd)*

*all :*

*# run kernel build system to make module*

*$(MAKE) -C $(BUILDSYSTEM\_DIR) M=$(PWD) modules*

*clean:*

*# run kernel build system to cleanup in current directory*

*$(MAKE) -C $(BUILDSYSTEM\_DIR) M=$(PWD) clean*

*load:*

*insmod ./$(TARGET\_MODULE).ko*

*unload:*

*rmmod ./$(TARGET\_MODULE).ko*

*endif*

The load target loads the build module and the unload target deletes it from the kernel.

In our tutorial, we’ve used code from main.c and device\_file.c to compile a driver. The resulting driver is named simple-module.ko. Let’s see how to use it.

**Loading and using the module**

To load the module, we have to execute the make load command from the source file folder. After this, the name of the driver is added to the /proc/modules file, while the device that the module registers is added to the /proc/devices file. The added records look like this:

*Character devices:*

*1 mem*

*4 tty*

*4 ttyS*

*…*

*250 Simple-driver*

*…*

The first three records contain the name of the added device and the major device number with which it’s associated. The minor number range allows device files to be created in the /dev virtual file system.

Then we need to create the special character file for our major number with the mknod /dev/simple-driver c 250 0 command.

After we’ve created the device file, we need to perform the final verification to make sure that what we’ve done works as expected. To verify, we can use the cat command to display the device file contents:

$> cat /dev/simple-driver

If we see the contents of our driver, it works correctly!

**Add device files under /dev**

Once again, we firstly provide the code, and then explain the example code.

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h> /\* printk() \*/

#include <linux/slab.h> /\* kmalloc() \*/

#include <linux/fs.h> /\* everything... \*/

#include <linux/errno.h> /\* error codes \*/

#include <linux/types.h> /\* size\_t \*/

#include <linux/fcntl.h> /\* O\_ACCMODE \*/

#include <linux/cdev.h>

#include <asm/uaccess.h> /\* copy\_\*\_user \*/

MODULE\_LICENSE("Dual BSD/GPL");

MODULE\_AUTHOR("Hcamael");

int scull\_major = 0;

int scull\_minor = 0;

int scull\_nr\_devs = 4;

int scull\_quantum = 4000;

int scull\_qset = 1000;

struct scull\_qset {

void \*\*data;

struct scull\_qset \*next;

};

struct scull\_dev {

struct scull\_qset \*data; /\* Pointer to first quantum set. \*/

int quantum; /\* The current quantum size. \*/

int qset; /\* The current array size. \*/

unsigned long size; /\* Amount of data stored here. \*/

unsigned int access\_key; /\* Used by sculluid and scullpriv. \*/

struct mutex mutex; /\* Mutual exclusion semaphore. \*/

struct cdev cdev; /\* Char device structure. \*/

};

struct scull\_dev \*scull\_devices; /\* allocated in scull\_init\_module \*/

/\*

\* Follow the list.

\*/

struct scull\_qset \*scull\_follow(struct scull\_dev \*dev, int n)

{

struct scull\_qset \*qs = dev->data;

/\* Allocate the first qset explicitly if need be. \*/

if (! qs) {

qs = dev->data = kmalloc(sizeof(struct scull\_qset), GFP\_KERNEL);

if (qs == NULL)

return NULL;

memset(qs, 0, sizeof(struct scull\_qset));

}

/\* Then follow the list. \*/

while (n--) {

if (!qs->next) {

qs->next = kmalloc(sizeof(struct scull\_qset), GFP\_KERNEL);

if (qs->next == NULL)

return NULL;

memset(qs->next, 0, sizeof(struct scull\_qset));

}

qs = qs->next;

continue;

}

return qs;

}

/\*

\* Data management: read and write.

\*/

ssize\_t scull\_read(struct file \*filp, char \_\_user \*buf, size\_t count,

loff\_t \*f\_pos)

{

struct scull\_dev \*dev = filp->private\_data;

struct scull\_qset \*dptr; /\* the first listitem \*/

int quantum = dev->quantum, qset = dev->qset;

int itemsize = quantum \* qset; /\* how many bytes in the listitem \*/

int item, s\_pos, q\_pos, rest;

ssize\_t retval = 0;

if (mutex\_lock\_interruptible(&dev->mutex))

return -ERESTARTSYS;

if (\*f\_pos >= dev->size)

goto out;

if (\*f\_pos + count > dev->size)

count = dev->size - \*f\_pos;

/\* Find listitem, qset index, and offset in the quantum \*/

item = (long)\*f\_pos / itemsize;

rest = (long)\*f\_pos % itemsize;

s\_pos = rest / quantum; q\_pos = rest % quantum;

/\* follow the list up to the right position (defined elsewhere) \*/

dptr = scull\_follow(dev, item);

if (dptr == NULL || !dptr->data || ! dptr->data[s\_pos])

goto out; /\* don't fill holes \*/

/\* read only up to the end of this quantum \*/

if (count > quantum - q\_pos)

count = quantum - q\_pos;

if (raw\_copy\_to\_user(buf, dptr->data[s\_pos] + q\_pos, count)) {

retval = -EFAULT;

goto out;

}

\*f\_pos += count;

retval = count;

out:

mutex\_unlock(&dev->mutex);

return retval;

}

ssize\_t scull\_write(struct file \*filp, const char \_\_user \*buf, size\_t count,

loff\_t \*f\_pos)

{

struct scull\_dev \*dev = filp->private\_data;

struct scull\_qset \*dptr;

int quantum = dev->quantum, qset = dev->qset;

int itemsize = quantum \* qset;

int item, s\_pos, q\_pos, rest;

ssize\_t retval = -ENOMEM; /\* Value used in "goto out" statements. \*/

if (mutex\_lock\_interruptible(&dev->mutex))

return -ERESTARTSYS;

/\* Find the list item, qset index, and offset in the quantum. \*/

item = (long)\*f\_pos / itemsize;

rest = (long)\*f\_pos % itemsize;

s\_pos = rest / quantum;

q\_pos = rest % quantum;

/\* Follow the list up to the right position. \*/

dptr = scull\_follow(dev, item);

if (dptr == NULL)

goto out;

if (!dptr->data) {

dptr->data = kmalloc(qset \* sizeof(char \*), GFP\_KERNEL);

if (!dptr->data)

goto out;

memset(dptr->data, 0, qset \* sizeof(char \*));

}

if (!dptr->data[s\_pos]) {

dptr->data[s\_pos] = kmalloc(quantum, GFP\_KERNEL);

if (!dptr->data[s\_pos])

goto out;

}

/\* Write only up to the end of this quantum. \*/

if (count > quantum - q\_pos)

count = quantum - q\_pos;

if (raw\_copy\_from\_user(dptr->data[s\_pos]+q\_pos, buf, count)) {

retval = -EFAULT;

goto out;

}

\*f\_pos += count;

retval = count;

/\* Update the size. \*/

if (dev->size < \*f\_pos)

dev->size = \*f\_pos;

out:

mutex\_unlock(&dev->mutex);

return retval;

}

/\* Beginning of the scull device implementation. \*/

/\*

\* Empty out the scull device; must be called with the device

\* mutex held.

\*/

int scull\_trim(struct scull\_dev \*dev)

{

struct scull\_qset \*next, \*dptr;

int qset = dev->qset; /\* "dev" is not-null \*/

int i;

for (dptr = dev->data; dptr; dptr = next) { /\* all the list items \*/

if (dptr->data) {

for (i = 0; i < qset; i++)

kfree(dptr->data[i]);

kfree(dptr->data);

dptr->data = NULL;

}

next = dptr->next;

kfree(dptr);

}

dev->size = 0;

dev->quantum = scull\_quantum;

dev->qset = scull\_qset;

dev->data = NULL;

return 0;

}

int scull\_release(struct inode \*inode, struct file \*filp)

{

printk(KERN\_DEBUG "process %i (%s) success release minor(%u) file\n", current->pid, current->comm, iminor(inode));

return 0;

}

/\*

\* Open and close

\*/

int scull\_open(struct inode \*inode, struct file \*filp)

{

struct scull\_dev \*dev; /\* device information \*/

dev = container\_of(inode->i\_cdev, struct scull\_dev, cdev);

filp->private\_data = dev; /\* for other methods \*/

/\* If the device was opened write-only, trim it to a length of 0. \*/

if ( (filp->f\_flags & O\_ACCMODE) == O\_WRONLY) {

if (mutex\_lock\_interruptible(&dev->mutex))

return -ERESTARTSYS;

scull\_trim(dev); /\* Ignore errors. \*/

mutex\_unlock(&dev->mutex);

}

printk(KERN\_DEBUG "process %i (%s) success open minor(%u) file\n", current->pid, current->comm, iminor(inode));

return 0;

}

/\*

\* The "extended" operations -- only seek.

\*/

loff\_t scull\_llseek(struct file \*filp, loff\_t off, int whence)

{

struct scull\_dev \*dev = filp->private\_data;

loff\_t newpos;

switch(whence) {

case 0: /\* SEEK\_SET \*/

newpos = off;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + off;

break;

case 2: /\* SEEK\_END \*/

newpos = dev->size + off;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if (newpos < 0)

return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

struct file\_operations scull\_fops = {

.owner = THIS\_MODULE,

.llseek = scull\_llseek,

.read = scull\_read,

.write = scull\_write,

// .unlocked\_ioctl = scull\_ioctl,

.open = scull\_open,

.release = scull\_release,

};

/\*

\* Set up the char\_dev structure for this device.

\*/

static void scull\_setup\_cdev(struct scull\_dev \*dev, int index)

{

int err, devno = MKDEV(scull\_major, scull\_minor + index);

cdev\_init(&dev->cdev, &scull\_fops);

dev->cdev.owner = THIS\_MODULE;

dev->cdev.ops = &scull\_fops;

err = cdev\_add (&dev->cdev, devno, 1);

/\* Fail gracefully if need be. \*/

if (err)

printk(KERN\_NOTICE "Error %d adding scull%d", err, index);

else

printk(KERN\_INFO "scull: %d add success\n", index);

}

void scull\_cleanup\_module(void)

{

int i;

dev\_t devno = MKDEV(scull\_major, scull\_minor);

/\* Get rid of our char dev entries. \*/

if (scull\_devices) {

for (i = 0; i < scull\_nr\_devs; i++) {

scull\_trim(scull\_devices + i);

cdev\_del(&scull\_devices[i].cdev);

}

kfree(scull\_devices);

}

/\* cleanup\_module is never called if registering failed. \*/

unregister\_chrdev\_region(devno, scull\_nr\_devs);

printk(KERN\_INFO "scull: cleanup success\n");

}

int scull\_init\_module(void)

{

int result, i;

dev\_t dev = 0;

/\*

\* Get a range of minor numbers to work with, asking for a dynamic major

\* unless directed otherwise at load time.

\*/

if (scull\_major) {

dev = MKDEV(scull\_major, scull\_minor);

result = register\_chrdev\_region(dev, scull\_nr\_devs, "scull");

} else {

result = alloc\_chrdev\_region(&dev, scull\_minor, scull\_nr\_devs, "scull");

scull\_major = MAJOR(dev);

}

if (result < 0) {

printk(KERN\_WARNING "scull: can't get major %d\n", scull\_major);

return result;

} else {

printk(KERN\_INFO "scull: get major %d success\n", scull\_major);

}

/\*

\* Allocate the devices. This must be dynamic as the device number can

\* be specified at load time.

\*/

scull\_devices = kmalloc(scull\_nr\_devs \* sizeof(struct scull\_dev), GFP\_KERNEL);

if (!scull\_devices) {

result = -ENOMEM;

goto fail;

}

memset(scull\_devices, 0, scull\_nr\_devs \* sizeof(struct scull\_dev));

/\* Initialize each device. \*/

for (i = 0; i < scull\_nr\_devs; i++) {

scull\_devices[i].quantum = scull\_quantum;

scull\_devices[i].qset = scull\_qset;

mutex\_init(&scull\_devices[i].mutex);

scull\_setup\_cdev(&scull\_devices[i], i);

}

return 0; /\* succeed \*/

fail:

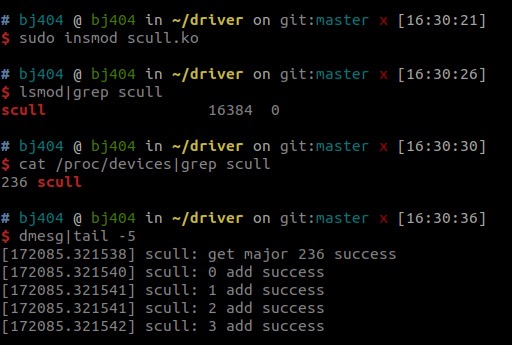
scull\_cleanup\_module();

return result;

}

module\_init(scull\_init\_module);

module\_exit(scull\_cleanup\_module);



**Conclusion**

We’ve shown you how to write a simple Linux driver. You can find the full source code of this driver in the Apriorit GitHub repository. If you need a more complex device driver, you may use this tutorial as a basis and add more functions and context to it.

At Apriorit, we’ve made Linux kernel and driver development our speciality. Our developers have successfully delivered hundreds of complex drivers for Linux, Unix, macOS, and Windows. Contact our experienced team to start working on your next Linux driver development project!

**Resources**

Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

The Linux Kernel Module Programming Guide by Peter Jay Salzman, and Ori Pomeranz

Linux Cross Reference