**chardev2.c**

/\*

 \* chardev2.c - Create an input/output character device

 \*/

#include <linux/cdev.h>

#include <linux/delay.h>

#include <linux/device.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/irq.h>

#include <linux/kernel.h> /\* We are doing kernel work \*/

#include <linux/module.h> /\* Specifically, a module \*/

#include <linux/poll.h>

#include "chardev.h"

#define SUCCESS 0

#define DEVICE\_NAME "char\_dev"

#define BUF\_LEN 80

enum {

    CDEV\_NOT\_USED = 0,

    CDEV\_EXCLUSIVE\_OPEN = 1,

};

/\* Is the device open right now? Used to prevent concurrent access into

 \* the same device

 \*/

static atomic\_t already\_open = ATOMIC\_INIT(CDEV\_NOT\_USED);

/\* The message the device will give when asked \*/

static char message[BUF\_LEN + 1];

static struct class \*cls;

/\* This is called whenever a process attempts to open the device file \*/

static int device\_open(struct inode \*inode, struct file \*file)

{

    pr\_info("device\_open(%p)\n", file);

    try\_module\_get(THIS\_MODULE);

    return SUCCESS;

}

static int device\_release(struct inode \*inode, struct file \*file)

{

    pr\_info("device\_release(%p,%p)\n", inode, file);

    module\_put(THIS\_MODULE);

    return SUCCESS;

}

/\* This function is called whenever a process which has already opened the

 \* device file attempts to read from it.

 \*/

static ssize\_t device\_read(struct file \*file, /\* see include/linux/fs.h \*/

    char \_\_user \*buffer, /\* buffer to be filled \*/

    size\_t length, /\* length of the buffer \*/

    loff\_t \*offset)

{

    /\* Number of bytes actually written to the buffer \*/

    int bytes\_read = 0;

    /\* How far did the process reading the message get? Useful if the message

    \* is larger than the size of the buffer we get to fill in device\_read.

    \*/

    const char \*message\_ptr = message;

    if (!\*(message\_ptr + \*offset)) { /\* we are at the end of message \*/

        \*offset = 0; /\* reset the offset \*/

        return 0; /\* signify end of file \*/

    }

    message\_ptr += \*offset;

    /\* Actually put the data into the buffer \*/

    while (length && \*message\_ptr) {

        /\* Because the buffer is in the user data segment, not the kernel

        \* data segment, assignment would not work. Instead, we have to

        \* use put\_user which copies data from the kernel data segment to

        \* the user data segment.

        \*/

        put\_user(\*(message\_ptr++), buffer++);

        length--;

        bytes\_read++;

    }

    pr\_info("Read %d bytes, %ld left\n", bytes\_read, length);

    \*offset += bytes\_read;

    /\* Read functions are supposed to return the number of bytes actually

    \* inserted into the buffer.

    \*/

    return bytes\_read;

}

/\* called when somebody tries to write into our device file. \*/

static ssize\_t device\_write(struct file \*file, const char \_\_user \*buffer,

size\_t length, loff\_t \*offset)

{

    int i;

    pr\_info("device\_write(%p,%p,%ld)", file, buffer, length);

    for (i = 0; i < length && i < BUF\_LEN; i++)

        get\_user(message[i], buffer + i);

    /\* Again, return the number of input characters used. \*/

    return i;

}

/\* This function is called whenever a process tries to do an ioctl on our

 \* device file. We get two extra parameters (additional to the inode and file

 \* structures, which all device functions get): the number of the ioctl

    called

 \* and the parameter given to the ioctl function.

 \*

 \* If the ioctl is write or read/write (meaning output is returned to the

 \* calling process), the ioctl call returns the output of this function.

 \*/

static long

device\_ioctl(struct file \*file, /\* ditto \*/

unsigned int ioctl\_num, /\* number and param for ioctl \*/

unsigned long ioctl\_param)

{

    int i;

    long ret = SUCCESS;

    /\* We don't want to talk to two processes at the same time. \*/

    if (atomic\_cmpxchg(&already\_open, CDEV\_NOT\_USED, CDEV\_EXCLUSIVE\_OPEN))

        return -EBUSY;

    /\* Switch according to the ioctl called \*/

    switch (ioctl\_num) {

    case IOCTL\_SET\_MSG: {

    /\* Receive a pointer to a message (in user space) and set that to

    \* be the device's message. Get the parameter given to ioctl by

    \* the process.

    \*/

        char \_\_user \*tmp = (char \_\_user \*)ioctl\_param;

        char ch;

        /\* Find the length of the message \*/

        get\_user(ch, tmp);

        for (i = 0; ch && i < BUF\_LEN; i++, tmp++)

            get\_user(ch, tmp);

            device\_write(file, (char \_\_user \*)ioctl\_param, i, NULL);

        break;

    }

    case IOCTL\_GET\_MSG: {

        loff\_t offset = 0;

    /\* Give the current message to the calling process - the parameter

    \* we got is a pointer, fill it.

    \*/

    i = device\_read(file, (char \_\_user \*)ioctl\_param, 99, &offset);

    /\* Put a zero at the end of the buffer, so it will be properly

    \* terminated.

    \*/

    put\_user('\0', (char \_\_user \*)ioctl\_param + i);

    break;

    }

    case IOCTL\_GET\_NTH\_BYTE:

    /\* This ioctl is both input (ioctl\_param) and output (the return

    \* value of this function).

    \*/

    ret = (long)message[ioctl\_param];

    break;

    }

    /\* We're now ready for our next caller \*/

    atomic\_set(&already\_open, CDEV\_NOT\_USED);

    return ret;

}

/\* Module Declarations \*/

/\* This structure will hold the functions to be called when a process does

 \* something to the device we created. Since a pointer to this structure

 \* is kept in the devices table, it can't be local to init\_module. NULL is

 \* for unimplemented functions.

 \*/

static struct file\_operations fops = {

    .read = device\_read,

    .write = device\_write,

    .unlocked\_ioctl = device\_ioctl,

    .open = device\_open,

    .release = device\_release, /\* a.k.a. close \*/

};

/\* Initialize the module - Register the character device \*/

static int \_\_init chardev2\_init(void)

{

    /\* Register the character device (atleast try) \*/

    int ret\_val = register\_chrdev(MAJOR\_NUM, DEVICE\_NAME, &fops);

    /\* Negative values signify an error \*/

    if (ret\_val < 0) {

        pr\_alert("%s failed with %d\n",

        "Sorry, registering the character device ", ret\_val);

        return ret\_val;

    }

    cls = class\_create(THIS\_MODULE, DEVICE\_FILE\_NAME);

    device\_create(cls, NULL, MKDEV(MAJOR\_NUM, 0), NULL, DEVICE\_FILE\_NAME);

    pr\_info("Device created on /dev/%s\n", DEVICE\_FILE\_NAME);

    return 0;

}

/\* Cleanup - unregister the appropriate file from /proc \*/

static void \_\_exit chardev2\_exit(void)

{

    device\_destroy(cls, MKDEV(MAJOR\_NUM, 0));

    class\_destroy(cls);

    /\* Unregister the device \*/

    unregister\_chrdev(MAJOR\_NUM, DEVICE\_NAME);

}

module\_init(chardev2\_init);

module\_exit(chardev2\_exit);

MODULE\_LICENSE("GPL");

chardev.h

/\*

 \* chardev.h - the header file with the ioctl definitions.

 \*

 \* The declarations here have to be in a header file, because they need

 \* to be known both to the kernel module (in chardev2.c) and the process

 \* calling ioctl() (in userspace\_ioctl.c).

 \*/

#ifndef CHARDEV\_H

#define CHARDEV\_H

#include <linux/ioctl.h>

/\* The major device number. We can not rely on dynamic registration

 \* any more, because ioctls need to know it.

 \*/

#define MAJOR\_NUM 100

/\* Set the message of the device driver \*/

#define IOCTL\_SET\_MSG \_IOW(MAJOR\_NUM, 0, char \*)

/\* \_IOW means that we are creating an ioctl command number for passing

 \* information from a user process to the kernel module.

 \*

 \* The first arguments, MAJOR\_NUM, is the major device number we are using.

 \*

 \* The second argument is the number of the command (there could be several

 \* with different meanings).

 \*

 \* The third argument is the type we want to get from the process to the

 \* kernel.

 \*/

/\* Get the message of the device driver \*/

#define IOCTL\_GET\_MSG \_IOR(MAJOR\_NUM, 1, char \*)

/\* This IOCTL is used for output, to get the message of the device driver.

 \* However, we still need the buffer to place the message in to be input,

 \* as it is allocated by the process.

 \*/

/\* Get the n'th byte of the message \*/

#define IOCTL\_GET\_NTH\_BYTE \_IOWR(MAJOR\_NUM, 2, int)

/\* The IOCTL is used for both input and output. It receives from the user

 \* a number, n, and returns message[n].

 \*/

 /\* The name of the device file \*/

#define DEVICE\_FILE\_NAME "char\_dev"

#define DEVICE\_PATH "/dev/char\_dev"

#endif

Makefile

obj-m += chardev2.o

PWD := $(CURDIR)

all:

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

clean:

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

userspace\_ioctl.c

/\* userspace\_ioctl.c - the process to use ioctl's to control the kernel

    module

  \*

  \* Until now we could have used cat for input and output. But now

  \* we need to do ioctl's, which require writing our own process.

\*/

/\* device specifics, such as ioctl numbers and the

 \* major device file. \*/

#include "./chardev.h"

#include <stdio.h> /\* standard I/O \*/

#include <fcntl.h> /\* open \*/

#include <unistd.h> /\* close \*/

#include <stdlib.h> /\* exit \*/

#include <sys/ioctl.h> /\* ioctl \*/

/\* Functions for the ioctl calls \*/

int ioctl\_set\_msg(int file\_desc, char \*message)

{

    int ret\_val;

    ret\_val = ioctl(file\_desc, IOCTL\_SET\_MSG, message);

    if (ret\_val < 0) {

        printf("ioctl\_set\_msg failed:%d\n", ret\_val);

    }

    return ret\_val;

}

int ioctl\_get\_msg(int file\_desc)

{

    int ret\_val;

    char message[100] = { 0 };

/\* Warning - this is dangerous because we don't tell

 \* the kernel how far it's allowed to write, so it

 \* might overflow the buffer. In a real production

 \* program, we would have used two ioctls - one to tell

 \* the kernel the buffer length and another to give

 \* it the buffer to fill

 \*/

    ret\_val = ioctl(file\_desc, IOCTL\_GET\_MSG, message);

    if (ret\_val < 0) {

        printf("ioctl\_get\_msg failed:%d\n", ret\_val);

    }

    printf("get\_msg message:%s", message);

    return ret\_val;

}

int ioctl\_get\_nth\_byte(int file\_desc)

{

    int i, c;

    printf("get\_nth\_byte message:");

    i = 0;

    do {

        c = ioctl(file\_desc, IOCTL\_GET\_NTH\_BYTE, i++);

        if (c < 0) {

            printf("\nioctl\_get\_nth\_byte failed at the %d'th byte:\n", i);

        return c;

        }

        putchar(c);

    } while (c != 0);

        return 0;

    }

/\* Main - Call the ioctl functions \*/

int main(void)

{

    int file\_desc, ret\_val;

    char \*msg = "Message passed by ioctl\n";

    file\_desc = open(DEVICE\_PATH, O\_RDWR);

    if (file\_desc < 0) {

        printf("Can't open device file: %s, error:%d\n", DEVICE\_PATH, file\_desc);

        exit(EXIT\_FAILURE);

    }

    ret\_val = ioctl\_set\_msg(file\_desc, msg);

    if (ret\_val)

        goto error;

    ret\_val = ioctl\_get\_nth\_byte(file\_desc);

    if (ret\_val)

        goto error;

    ret\_val = ioctl\_get\_msg(file\_desc);

    if (ret\_val)

        goto error;

    close(file\_desc);

    return 0;

error:

    close(file\_desc);

    exit(EXIT\_FAILURE);

 }

dmesg result

Text

Description automatically generated