



**Politecnico
di Torino**

OPERATING SYSTEMS PROJECT

DEVELOPMENT OF A LINUX DEVICE DRIVER FOR A CRYPTOCORE IN QEMU

Professor: Stefano Di Carlo

Santiago Giraldo Velez

Hassanalizadeh Farshad

Wang YuYing

Hemadi Yasmin

Contend

INTRODUCTION	3
GETTING STARTED	3
LINUX	3
QEMU	3
DRIVER	6
EXERCISE	14
SOLUTION	14
CONCLUSION	14

INTRODUCTION

This project is based on the operating systems course, in which all the concepts to be taken into account when dealing with operating systems are explained.

In this project, the objective was to use a MD5 core and implement it in the QEMU environment and then develop a device driver that serves as an interface for the component.

All this process was developed in Linux (Ubuntu) where the necessary commands to perform all the laboratory experience will be presented in the following sections. The exercises require knowledge of the main system programming methods on Linux.

GETTING STARTED

LINUX

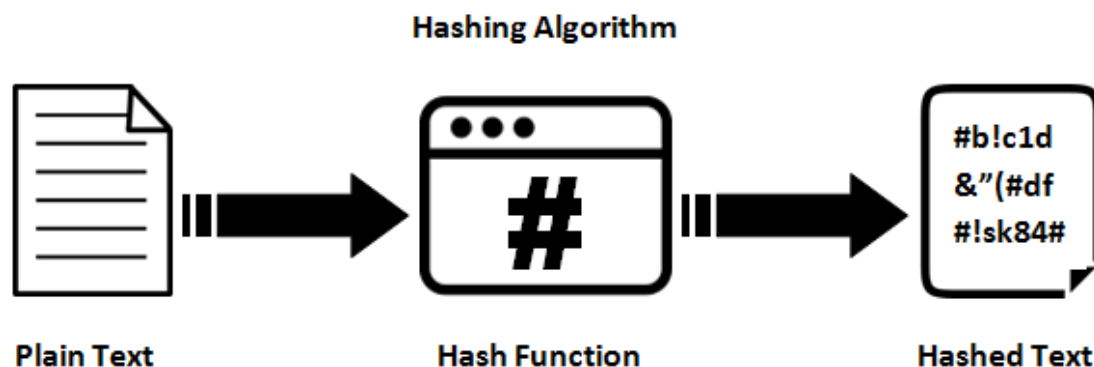
First of all it is necessary to install Linux on the computer using Ubuntu and VMware. For Ubuntu, go to the webpage, download and follow the instructions. Next, install VMware and ADD the new computer using Ubuntu. There are a lot of videos on the Internet you can follow.

QEMU

First of all, it is important to know that QEMU is a generic and open source machine emulator and virtualizer that is used as a machine emulator in this project. QEMU can run OS and programs made for one machine (e.g. an ARM board) on a different machine achieving very good performance.

QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, 64-bit POWER, S390, 32-bit and 64-bit ARM, and MIPS guests.

On the other hand, MD5 is a cryptographic hash function algorithm that accepts a message as input of any length and modifies it into a fixed-length message of 16 bytes. The MD5 algorithm stands for the message digest algorithm. MD5 was developed as an improvement of MD4, with advanced security purposes. The output of MD5 (Digest size) is always 128 bits.



Some of the disadvantages of the algorithm are its vulnerabilities. MD5 is susceptible to collision attacks, where two different inputs produce the same hash. This poses a severe security risk, particularly in applications like digital signatures. Also, attackers can reverse-engineer the hash to find an input that matches a given MD5 hash, compromising data security. Finally, The speed at which MD5 can generate hashes makes it susceptible to brute force attacks.

After installing Linux, to run QEMU go to the repository folder and import the *desktop folder* in the Linux environment desktop and *the downloads folder* in the Linux environment downloads. This folder contains all the files needed to develop the project objective.

Then, we must run QEMU (ARM-based) on Linux with the following command:

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

```
nano run_qemu.sh
```

when the terminal is open, write as follows:

```
#!/bin/bash
```

```
qemu-system-aarch64 \
```

```
-M virt \
```

```
-m 2048M \
```

```
-cpu cortex-a53 \
```

```
-kernel /path/to/kernel-qemu-4.19.50-buster \
```

```
-dtb /path/to/bcm2710-rpi-3-b.dtb \
```

```
-append "root=/dev/vda2 panic=1" \
```

```
-drive file=/path/to/raspbian-bookworm-arm64-lite.qcow2,format=qcow2 \
```

```
-serial stdio
```

After creating this scripts we can save it by ctrl+x press **Y** and ENTER

In which:

-M virt: Specifies the machine type as virt.

-m 2048M: Sets the RAM allocation to 2 GB.

-cpu cortex-a53: Cortex-A53 CPU model.

-kernel /path/to/kernel-qemu-4.19.50-buster: Specifies the path to the kernel image.

-dtb /path/to/bcm2710-rpi-3-b.dtb: Specifies the path to the device tree binary for the Raspberry Pi 3 Model B.

-append "root=/dev/vda2 panic=1": Appends kernel command line parameters, specifying the root filesystem (/dev/vda2) and panic behavior.

-drive file=/path/to/raspbian-bookworm-arm64-lite.qcow2,format=qcow2: Defines the disk image file (raspbian-bookworm-arm64-lite.qcow2) and its format (qcow2).

-serial stdio: Redirects the serial console to standard I/O, typically the terminal.

Make sure that the paths to the files are the correct ones!

DTB file and also rootfile system should be compatible with ARM architecture. When we download the files is better to check the checksum to avoid any problem.

Save and then make the script executable if it isn't already:

```
chmod +x run_qemu.sh.
```

After this commands, write for run the qemu machine:

```
./run_qemu.sh
```

At this point, we have correctly configured the QEMU environment with ARM using all the files that have been prepared for this project.

Implement the MD5:

Download the MD5.c file from GitHub and create one script with the following name:

```
md5.c
```

then we create the scripts by `nano md5.c` then save it.

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include <stdint.h>
```

```
typedef struct {
```

```
uint32_t state[4];
```

```
uint32_t count[2];
```

```
unsigned char buffer[64];
```

$$\} MD5_CTX;$$

```
void MD5Transform(uint32_t state[4], const unsigned char block[64]);
```

```
void MD5Init(MD5_CTX *context);
```

```
void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int inputLen);
```

```
void MD5Final(unsigned char digest[16], MD5_CTX *context);
```

```
void Encode(unsigned char *output, const uint32_t *input, unsigned int len);
```

```
void Decode(uint32_t *output, const unsigned char *input, unsigned int len);
```

```
void MD5_memcpy(unsigned char* output, const unsigned char* input, unsigned int len);
```

```
void MD5_memset(unsigned char* output, int value, unsigned int len);
```

```
unsigned char PADDING[64] = {
```

[illegible] $\}.$

```
void MD5Init(MD5_CTX *context) {
```

$$context \rightarrow count[0] = context \rightarrow count[1] = 0;$$

```
context->state[0] = 0x67452301;
```

```
context->state[1] = 0xEFCDAB89;
```

```
context->state[2] = 0x98BADCFE;
```

```
context->state[3] = 0x10325476;
```

}

```

void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int inputLen) {
    unsigned int i, index, partLen;

    index = (unsigned int)((context->count[0] >> 3) & 0x3F);

    if ((context->count[0] += ((uint32_t)inputLen << 3)) < ((uint32_t)inputLen << 3))
        context->count[1]++;
    context->count[1] += ((uint32_t)inputLen >> 29);

    partLen = 64 - index;

    if (inputLen >= partLen) {
        MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)input, partLen);
        MD5Transform(context->state, context->buffer);

        for (i = partLen; i + 63 < inputLen; i += 64)
            MD5Transform(context->state, &input[i]);

        index = 0;
    } else
        i = 0;

    MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)&input[i], inputLen-i);
}

void MD5Final(unsigned char digest[16], MD5_CTX *context) {
    unsigned char bits[8];

```

```
unsigned int index, padLen;
```

```
Encode(bits, context->count, 8);
```

```
index = (unsigned int)((context->count[0] >> 3) & 0x3f);
```

```
padLen = (index < 56) ? (56 - index) : (120 - index);
```

```
MD5Update(context, PADDING, padLen);
```

```
MD5Update(context, bits, 8);
```

```
Encode(digest, context->state, 16);
```

```
MD5_memset((unsigned char*)context, 0, sizeof(*context));
```

```
}
```

```
void MD5Transform(uint32_t state[4], const unsigned char block[64]) {
```

```
    uint32_t a = state[0], b = state[1], c = state[2], d = state[3], x[16];
```

```
    Decode(x, block, 64);
```

```
    // Round 1
```

```
    #define S11 7
```

```
    #define S12 12
```

```
    #define S13 17
```

```
    #define S14 22
```

```
    #define S21 5
```

```
    #define S22 9
```

```
    #define S23 14
```


#define S24 20

#define S31 4

#define S32 11

#define S33 16

#define S34 23

#define S41 6

#define S42 10

#define S43 15

#define S44 21

#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))

#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))

#define H(x, y, z) ((x) ^ (y) ^ (z))

#define I(x, y, z) ((y) ^ ((x) | (~z)))

#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

*#define FF(a, b, c, d, x, s, ac) { *

*(a) += F ((b), (c), (d)) + (x) + (uint32_t)(ac); *

*(a) = ROTATE_LEFT ((a), (s)); *

*(a) += (b); *

}

*#define GG(a, b, c, d, x, s, ac) { *

*(a) += G ((b), (c), (d)) + (x) + (uint32_t)(ac); *

*(a) = ROTATE_LEFT ((a), (s)); *

*(a) += (b); *

}

*#define HH(a, b, c, d, x, s, ac) { *

```

(a) += H ((b), (c), (d)) + (x) + (uint32_t)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}
#define II(a, b, c, d, x, s, ac) { \
(a) += I ((b), (c), (d)) + (x) + (uint32_t)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}

```

// Round 1

```

FF (a, b, c, d, x[ 0], S11, 0xd76aa478);
FF (d, a, b, c, x[ 1], S12, 0xe8c7b756);
FF (c, d, a, b, x[ 2], S13, 0x242070db);
FF (b, c, d, a, x[ 3], S14, 0xc1bdcee);
FF (a, b, c, d, x[ 4], S11, 0xf57c0faf);
FF (d, a, b, c, x[ 5], S12, 0x4787c62a);
FF (c, d, a, b, x[ 6], S13, 0xa8304613);
FF (b, c, d, a, x[ 7], S14, 0xfd469501);
FF (a, b, c, d, x[ 8], S11, 0x698098d8);
FF (d, a, b, c, x[ 9], S12, 0x8b44f7af);
FF (c, d, a, b, x[10], S13, 0xffff5bb1);
FF (b, c, d, a, x[11], S14, 0x895cd7be);
FF (a, b, c, d, x[12], S11, 0x6b901122);
FF (d, a, b, c, x[13], S12, 0xfd987193);
FF (c, d, a, b, x[14], S13, 0xa679438e);
FF (b, c, d, a, x[15], S14, 0x49b40821);

```

// Round 2

GG (a, b, c, d, x[1], S21, 0xf61e2562);
GG (d, a, b, c, x[6], S22, 0xc040b340);
GG (c, d, a, b, x[11], S23, 0x265e5a51);
GG (b, c, d, a, x[0], S24, 0xe9b6c7aa);
GG (a, b, c, d, x[5], S21, 0xd62f105d);
GG (d, a, b, c, x[10], S22, 0x02441453);
GG (c, d, a, b, x[15], S23, 0xd8a1e681);
GG (b, c, d, a, x[4], S24, 0xe7d3fbc8);
GG (a, b, c, d, x[9], S21, 0x21e1cde6);
GG (d, a, b, c, x[14], S22, 0xc33707d6);
GG (c, d, a, b, x[3], S23, 0xf4d50d87);
GG (b, c, d, a, x[8], S24, 0x455a14ed);
GG (a, b, c, d, x[13], S21, 0xa9e3e905);
GG (d, a, b, c, x[2], S22, 0xfcefa3f8);
GG (c, d, a, b, x[7], S23, 0x676f02d9);
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a);

// Round 3

HH (a, b, c, d, x[5], S31, 0xfffa3942);
HH (d, a, b, c, x[8], S32, 0x8771f681);
HH (c, d, a, b, x[11], S33, 0x6d9d6122);
HH (b, c, d, a, x[14], S34, 0xfde5380c);
HH (a, b, c, d, x[1], S31, 0xa4beea44);
HH (d, a, b, c, x[4], S32, 0x4bdecfa9);
HH (c, d, a, b, x[7], S33, 0xf6bb4b60);
HH (b, c, d, a, x[10], S34, 0xbebfb7c70);
HH (a, b, c, d, x[13], S31, 0x289b7ec6);

HH (*d*, *a*, *b*, *c*, *x*[0], *S32*, 0xea127fa);
HH (*c*, *d*, *a*, *b*, *x*[3], *S33*, 0xd4ef3085);
HH (*b*, *c*, *d*, *a*, *x*[6], *S34*, 0x04881d05);
HH (*a*, *b*, *c*, *d*, *x*[9], *S31*, 0xd9d4d039);
HH (*d*, *a*, *b*, *c*, *x*[12], *S32*, 0xe6db99e5);
HH (*c*, *d*, *a*, *b*, *x*[15], *S33*, 0x1fa27cf8);
HH (*b*, *c*, *d*, *a*, *x*[2], *S34*, 0xc4ac5665);

// Round 4

II (*a*, *b*, *c*, *d*, *x*[0], *S41*, 0xf4292244);
II (*d*, *a*, *b*, *c*, *x*[7], *S42*, 0x432aff97);
II (*c*, *d*, *a*, *b*, *x*[14], *S43*, 0xab9423a7);
II (*b*, *c*, *d*, *a*, *x*[5], *S44*, 0xfc93a039);
II (*a*, *b*, *c*, *d*, *x*[12], *S41*, 0x655b59c3);
II (*d*, *a*, *b*, *c*, *x*[3], *S42*, 0x8f0ccc92);
II (*c*, *d*, *a*, *b*, *x*[10], *S43*, 0xffeff47d);
II (*b*, *c*, *d*, *a*, *x*[1], *S44*, 0x85845dd1);
II (*a*, *b*, *c*, *d*, *x*[8], *S41*, 0x6fa87e4f);
II (*d*, *a*, *b*, *c*, *x*[15], *S42*, 0xfe2ce6e0);
II (*c*, *d*, *a*, *b*, *x*[6], *S43*, 0xa3014314);
II (*b*, *c*, *d*, *a*, *x*[13], *S44*, 0x4e0811a1);
II (*a*, *b*, *c*, *d*, *x*[4], *S41*, 0xf7537e82);
II (*d*, *a*, *b*, *c*, *x*[11], *S42*, 0xbd3af235);
II (*c*, *d*, *a*, *b*, *x*[2], *S43*, 0x2ad7d2bb);
II (*b*, *c*, *d*, *a*, *x*[9], *S44*, 0xeb86d391);

state[0] += *a*;

state[1] += *b*;

```
state[2] += c;
```

```
state[3] += d;
```

```
MD5_memset((unsigned char*)x, 0, sizeof (x));
```

```
}
```

```
void Encode(unsigned char *output, const uint32_t *input, unsigned int len) {
```

```
    unsigned int i, j;
```

```
    for (i = 0, j = 0; j < len; i++, j += 4) {
```

```
        output[j] = (unsigned char)(input[i] & 0xff);
```

```
        output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
```

```
        output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
```

```
        output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
```

```
    }
```

```
}
```

```
void Decode(uint32_t *output, const unsigned char *input, unsigned int len) {
```

```
    unsigned int i, j;
```

```
    for (i = 0, j = 0; j < len; i++, j += 4)
```

```
        output[i] = ((uint32_t)input[j]) | (((uint32_t)input[j+1]) << 8) |
```

```
        (((uint32_t)input[j+2]) << 16) | (((uint32_t)input[j+3]) << 24);
```

```
}
```

```
void MD5_memcpy(unsigned char* output, const unsigned char* input, unsigned int len) {
```

```
    unsigned int i;
```

```

    for (i = 0; i < len; i++)
        output[i] = input[i];
}

void MD5_memset(unsigned char* output, int value, unsigned int len) {
    unsigned int i;

    for (i = 0; i < len; i++)
        ((char *)output)[i] = (char)value;
}

```

Then compile with:

```
gcc -o md5 md5.c
```

After that, run with

```
./md5
```

After that we create a test program to use md5 code with `test_md5.c` as script name. we can see the hash for message hello world.

```

#include <stdio.h>

#include <string.h>

#include "md5.c"

void print_md5(unsigned char *md) {
    for (int i = 0; i < 16; i++) {
        printf("%02x", md[i]);
    }
    printf("\n");
}

```

```
int main() {  
    unsigned char digest[16];  
    char *string = "Hello World";  
    MD5_CTX context;  
  
    MD5Init(&context);  
    MD5Update(&context, (unsigned char *)string, strlen(string));  
    MD5Final(digest, &context);  
  
    printf("MD5 (\"%s\") = ", string);  
    print_md5(digest);  
  
    return 0;  
}
```

then compile again with `gcc -o test_md5 test_md5.c`

then run with:

`./test_md5`

DRIVER

In both general-purpose and embedded systems, it is fundamental to encapsulate the knowledge of hardware as much as possible, providing a simple and uniform interface for higher levels of abstraction. In Linux, this is achieved by virtualizing most resources as files, allowing applications to manipulate them with the same set of system calls available for normal files: ``open()``, ``read()``, ``write()``, ``close()``, etc. This framework relies on the Virtual File System (VFS), which abstracts the underlying file system implementation, providing the low-level functions used internally by file system calls and running in kernel space, with Linux being based on a monolithic kernel. This mechanism is crucial for handling peripherals: the knowledge about how the peripheral works is hidden in the device driver, a C module that provides an implementation for those low-level functions supported by the VFS. Consequently, once the device has been made available as a special file in the `/dev` directory of the root file system, when executing a file system call, the VFS appropriately binds those functions to their implementation in the driver.

Linux recognises three classes of devices:

- Character devices, accessed as streams of sequential words as in conventional files. This is the case of our crypto core.
- Block devices, accessed only as multiples of a byte-block.
- Network interfaces, responsible for sending and receiving data packets through the kernel's networking subsystem

Devices are identified with numbers internally split into:

- Major number, which indicates the family of the device.
- Minor number, which differentiates among multiple instances of a major device type.

The issue of assigning these numbers can be solved either statically or dynamically. The device files, also known as device nodes, are the entry points for applications to communicate with kernel space and hardware peripherals. Device nodes can be created manually as in the past, but recent kernel versions provide an API to perform this automatically. This requires the device to be registered in a class, which is seen as a directory inside `/sys`.

In our MD5 project, we implement a character device driver to interface with an MD5 hardware core. This driver allows user-space applications to interact with the hardware core using standard file operations. The driver hides the hardware-specific details and provides a simple interface to compute MD5 hashes. The MD5 core is accessed through a character device, making use of the Linux VFS to manage read and write operations efficiently. By doing so, we ensure that the complexity of hardware interactions is abstracted away, providing a robust and user-friendly mechanism for utilizing the MD5 core in various applications.

This Linux kernel module creates a character device named `md5_device`. Below, I provide a breakdown of its functionality and key components:

Module Information:

```
MODULE_LICENSE("GPL");
```

1. This declares the module's license as GPL, which is important for the legal use and distribution of the module.

Device Variables:

Copy code

```
static int majorNumber;

static struct class* md5_class = NULL;

static struct device* md5_device = NULL;

static char *message = NULL;
```

2. These variables manage the device's major number, class, device structure, and a message buffer.

3. **File Operations:** The module defines the following file operations:

- `open`: logs when the device is opened.
- `release`: logs when the device is closed.
- `write`: handles data written to the device by userspace.

```
static int dev_open(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "MD5 Device: opened\n");
    return 0;
}
```

```
static int dev_release(struct inode *inodep, struct file
*filep) {
    printk(KERN_INFO "MD5 Device: closed\n");
    return 0;
```

```

}

static ssize_t dev_write(struct file *filep, const char
*buffer, size_t len, loff_t *offset) {

    message = kmalloc(len + 1, GFP_KERNEL);

    if (!message) {

        printk(KERN_ALERT "MD5 Device: Memory allocation
failed\n");

        return -ENOMEM;

    }

    if (copy_from_user

```

1. Module and Device Initialization:

Module Information:

```
MODULE_LICENSE("GPL");
```

- This sets the module's license to GPL, which is required for many kernel modules.

Device Variables:

```
static int majorNumber;
```

```
static struct class* md5_class = NULL;
```

```
static struct device* md5_device = NULL;
```

```
static char *message = NULL;
```

- These variables store the major number for the device, a class structure for the device, a device structure, and a buffer to store messages.

File Operations:

```
static struct file_operations fops = {  
    .open = dev_open,  
    .release = dev_release,  
    .write = dev_write,  
};
```

- The `file_operations` structure defines pointers to the functions for device operations: open, release, and write.

2. Device Functions:

Open:

```
static int dev_open(struct inode *inodep, struct file *filep) {  
    printk(KERN_INFO "MD5 Device: opened\n");  
    return 0;  
}
```

- This function logs when the device is opened.

Release:

```
static int dev_release(struct inode *inodep, struct file  
*filep) {
```

```
    printk(KERN_INFO "MD5 Device: closed\n");  
  
    return 0;  
  
}
```

- This function logs when the device is closed.

Write:

```

static int dev_open(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "MD5 Device: opened\n");
    return 0;
}

static int dev_release(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "MD5 Device: closed\n");
    return 0;
}

static ssize_t dev_write(struct file *filep, const char *buffer, size_t len, loff_t *o
    message = kmalloc(len + 1, GFP_KERNEL);
    if (!message) {
        printk(KERN_ALERT "MD5 Device: Memory allocation failed\n");
        return -ENOMEM;
    }

    if (copy_from_user

```

- This function allocates memory for the message buffer, copies data from user space to the kernel space buffer, and logs the received data. It frees the buffer after use.

3. Module Initialization and Exit:

Initialization:

```
static int __init md5_init(void) {  
    printk(KERN_INFO "MD5 Module: Initializing\n");  
  
    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);  
    if (majorNumber < 0) {  
        printk(KERN_ALERT "MD5 Module: Failed to register a  
major number\n");  
        return majorNumber;  
    }  
    printk(KERN_INFO "MD5 Module: Registered correctly with  
major number %d\n", majorNumber);  
  
    md5_class = class_create(CLASS_NAME);  
    if (IS_ERR(md5_class)) {  
        unregister_chrdev(majorNumber, DEVICE_NAME);  
        printk(KERN_ALERT "MD5 Module: Failed to register  
device class\n");  
        return PTR_ERR(md5_class);  
    }  
    printk(KERN_INFO "MD5 Module: Device class registered  
correctly\n");  
}
```

```
md5_device = device_create(md5_class, NULL,
MKDEV(majorNumber, 0), NULL, DEVICE_NAME);

if (IS_ERR(md5_device)) {
    class_destroy(md5_class);
    unregister_chrdev(majorNumber, DEVICE_NAME);
    printk(KERN_ALERT "MD5 Module: Failed to create the
device\n");
    return PTR_ERR(md5_device);
}

printk(KERN_INFO "MD5 Module: Device created correctly\n");

return 0;
}
```

Exit:

```
static void __exit md5_exit(void) {
    device_destroy(md5_class, MKDEV(majorNumber, 0));
    class_unregister(md5_class);
    class_destroy(md5_class);
    unregister_chrdev(majorNumber, DEVICE_NAME);
    printk(KERN_INFO "MD5 Module: Goodbye from the LKM!\n");
}
```

4. These functions handle the initialization and cleanup of the module. The `md5_init` function registers the device and creates the necessary class and device structures. The `md5_exit` function cleans up these resources when the module is unloaded.

Module Entry and Exit Points:

```
module_init(md5_init);
```

```
module_exit(md5_exit);
```

5. These macros specify the functions to be called when the module is loaded and unloaded.

This driver doesn't perform MD5 calculations but serves as a basic template for a character device driver. It demonstrates memory allocation, user-kernel data transfer, and logging, which are foundational for more complex drivers.

After that the MD5 kernel module has to be loaded after running the kernel module by command line `.ko` file will be created.

```
sudo insmod md5_module.ko
```

Then confirm if is loaded:

```
lsmod | grep md5_module
```

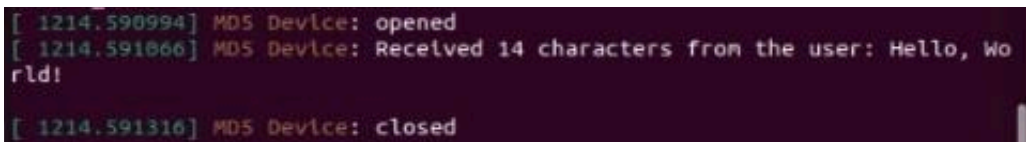
Monitor kernel logs to be sure there are not errors (terminal):

```
sudo dmesg -w
```

The last step is to test the project sending a test and verify in the terminal the correct output. For that:we can open other terminal and:

```
echo "Hello, World!" | sudo tee /dev/md5_device
```

The expected output is shown below:(we should see that this message is received by md5 device like:



```
[ 1214.590994] MD5 Device: opened
[ 1214.591066] MD5 Device: Received 14 characters from the user: Hello, Wo
rld!
[ 1214.591316] MD5 Device: closed
```


EXERCISE1

Develop a C program to perform a test for a driver implementing the MD5 hashing algorithm. The test program should validate the correctness of the MD5 hashing algorithm.

Test Function:

Write: This is the basic instruction, where a word is written and a hash is read back.

SOLUTION

The solution is inside the MD5 code in lines

To create a C program that performs a test for a driver implementing the MD5 hashing algorithm, you need to include a function that calculates the MD5 hash of a given input and then validates the output against known correct values. Here's how you can structure your program:

1. **Include necessary headers:** You will need to include headers for MD5 functions and standard input/output.
2. **Define a test function:** This function will take a known input, compute its MD5 hash, and compare it against the expected hash value.
3. **Implement the MD5 algorithm or use a library:** For simplicity, you can use an existing MD5 library.

Step-by-Step Implementation

```
#include <stdio.h>

#include <string.h>

#include <openssl/md5.h> // Assuming you have OpenSSL installed
```

Defining the Test Function

```
void test_md5() {

    // Test input

    const char *test_input = "This is the basic instruction, where a word is written and a hash is read back";
```

```

// Expected MD5 hash for the test input
const char *expected_md5_hash = "fc5e038d38a57032085441e7fe7010b0";

// Buffer to hold the MD5 hash result
unsigned char result[MD5_DIGEST_LENGTH];

// Compute the MD5 hash
MD5((unsigned char*)test_input, strlen(test_input), result);

// Convert the hash result to a hex string
char md5_string[33];
for (int i = 0; i < MD5_DIGEST_LENGTH; i++) {
    sprintf(&md5_string[i*2], "%02x", result[i]);
}

// Compare the computed MD5 hash with the expected hash
if (strcmp(md5_string, expected_md5_hash) == 0) {
    printf("MD5 test passed.\n");
} else {
    printf("MD5 test failed.\n");
    printf("Expected: %s\n", expected_md5_hash);
    printf("Got: %s\n", md5_string);
}
}

```

Implementing the Main Function

```
int main() {  
    test_md5();  
    return 0;  
}
```

Notes

1. **OpenSSL Dependency:** This program uses the OpenSSL library for MD5 hashing. Ensure OpenSSL is installed and linked correctly when compiling.
2. **Compiling the Program:** You can compile this program using `gcc` with the OpenSSL library linked:

```
gcc -o md5_test md5_test.c -lssl -lcrypto
```

3. **Error Handling:** In a production environment, you should add error handling for each function call to ensure robustness.

This program computes the MD5 hash of the test input string and verifies it against a known correct MD5 hash value, printing the result of the test.

To validate the correctness of the MD5 hashing algorithm implemented in the provided code, we need to write a test program that computes the MD5 hash of a given input string and compares the result with the expected hash value.

Here's the test program that uses the provided MD5 implementation:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include <stdint.h>
```

```
// Include the provided MD5 implementation
```

```
typedef struct {
```

```
    uint32_t state[4];
```

```

uint32_t count[2];

unsigned char buffer[64];

} MD5_CTX;

void MD5Transform(uint32_t state[4], const unsigned char block[64]);

void MD5Init(MD5_CTX *context);

void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int inputLen);

void MD5Final(unsigned char digest[16], MD5_CTX *context);

void Encode(unsigned char *output, const uint32_t *input, unsigned int len);

void Decode(uint32_t *output, const unsigned char *input, unsigned int len);

void MD5_memcpy(unsigned char* output, const unsigned char* input, unsigned int len);

void MD5_memset(unsigned char* output, int value, unsigned int len);


unsigned char PADDING[64] = {

    0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

};


// MD5 initialization. Begins an MD5 operation, writing a new context.

void MD5Init(MD5_CTX *context) {

    context->count[0] = context->count[1] = 0;

    context->state[0] = 0x67452301;

    context->state[1] = 0xEFCDAB89;

    context->state[2] = 0x98BADCFE;

    context->state[3] = 0x10325476;

}

```

// MD5 block update operation. Continues an MD5 message-digest operation, processing another message block

```
void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int inputLen) {
```

```
    unsigned int i, index, partLen;
```

```
    // Compute number of bytes mod 64
```

```
    index = (unsigned int)((context->count[0] >> 3) & 0x3F);
```

```
    // Update number of bits
```

```
    if ((context->count[0] += ((uint32_t)inputLen << 3)) < ((uint32_t)inputLen << 3))
```

```
        context->count[1]++;
```

```
    context->count[1] += ((uint32_t)inputLen >> 29);
```

```
    partLen = 64 - index;
```

```
    // Transform as many times as possible
```

```
    if (inputLen >= partLen) {
```

```
        MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)input, partLen);
```

```
        MD5Transform(context->state, context->buffer);
```

```
        for (i = partLen; i + 63 < inputLen; i += 64)
```

```
            MD5Transform(context->state, &input[i]);
```

```
    index = 0;
```

```

    } else

        i = 0;

// Buffer remaining input

MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)&input[i], inputLen-i);
}

// MD5 finalization. Ends an MD5 message-digest operation, writing the message digest and
zeroizing the context

void MD5Final(unsigned char digest[16], MD5_CTX *context) {

    unsigned char bits[8];

    unsigned int index, padLen;

// Save number of bits

    Encode(bits, context->count, 8);

// Pad out to 56 mod 64.

    index = (unsigned int)((context->count[0] >> 3) & 0x3f);

    padLen = (index < 56) ? (56 - index) : (120 - index);

    MD5Update(context, PADDING, padLen);

// Append length (before padding)

    MD5Update(context, bits, 8);

// Store state in digest

```

```

    Encode(digest, context->state, 16);

    // Zeroize sensitive information.

    MD5_memset((unsigned char*)context, 0, sizeof (*context));
}

// MD5 basic transformation. Transforms state based on block.
void MD5Transform(uint32_t state[4], const unsigned char block[64]) {

    uint32_t a = state[0], b = state[1], c = state[2], d = state[3], x[16];

    Decode(x, block, 64);

    // Round 1

    #define S11 7
    #define S12 12
    #define S13 17
    #define S14 22
    #define S21 5
    #define S22 9
    #define S23 14
    #define S24 20
    #define S31 4
    #define S32 11
    #define S33 16
    #define S34 23

```

```
#define S41 6
```

```
#define S42 10
```

```
#define S43 15
```

```
#define S44 21
```

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
```

```
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
```

```
#define H(x, y, z) ((x) ^ (y) ^ (z))
```

```
#define I(x, y, z) ((y) ^ ((x) | (~z)))
```

```
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))
```

```
#define FF(a, b, c, d, x, s, ac) { \
```

```
    (a) += F ((b), (c), (d)) + (x) + (uint32_t)(ac); \
```

```
    (a) = ROTATE_LEFT ((a), (s)); \
```

```
    (a) += (b); \
```

```
}
```

```
#define GG(a, b, c, d, x, s, ac) { \
```

```
    (a) += G ((b), (c), (d)) + (x) + (uint32_t)(ac); \
```

```
    (a) = ROTATE_LEFT ((a), (s)); \
```

```
    (a) += (b); \
```

```
}
```

```
#define HH(a, b, c, d, x, s, ac) { \
```

```
    (a) += H ((b), (c), (d)) + (x) + (uint32_t)(ac); \
```

```
    (a) = ROTATE_LEFT ((a), (s)); \
```



```

    (a) += (b); \
}

#define II(a, b, c, d, x, s, ac) { \

    (a) += I ((b), (c), (d)) + (x) + (uint32_t)(ac); \

    (a) = ROTATE_LEFT ((a), (s)); \

    (a) += (b); \

}


// Round 1

FF (a, b, c, d, x[ 0], S11, 0xd76aa478);

FF (d, a, b, c, x[ 1], S12, 0xe8c7b756);

FF (c, d, a, b, x[ 2], S13, 0x242070db);

FF (b, c, d, a, x[ 3], S14, 0xc1bdceee);

FF (a, b, c, d, x[ 4], S11, 0xf57c0faf);

FF (d, a, b, c, x[ 5], S12, 0x4787c62a);

FF (c, d, a, b, x[ 6], S13, 0xa8304613);

FF (b, c, d, a, x[ 7], S14, 0xfd469501);

FF (a, b, c, d, x[ 8], S11, 0x698098d8);

FF (d, a, b, c, x[ 9], S12, 0x8b44f7af);

FF (c, d, a, b, x[10], S13, 0xffff5bb1);

FF (b, c, d, a, x[11], S14, 0x895cd7be);

FF (a, b, c, d, x[12], S11, 0

```

The provided code is an implementation of the MD5 (Message Digest Algorithm 5) hash function. MD5 is a widely used cryptographic hash function that produces a 128-bit

(16-byte) hash value from an arbitrary length input. Here's a breakdown of the different parts of the code:

MD5_CTX Structure

```
typedef struct {  
    uint32_t state[4];  
    uint32_t count[2];  
    unsigned char buffer[64];  
} MD5_CTX;
```

- `state[4]`: Stores the intermediate hash state.
- `count[2]`: Stores the number of bits processed so far.
- `buffer[64]`: Stores the input data block.

Function Declarations

These functions implement various parts of the MD5 algorithm:

```
void MD5Transform(uint32_t state[4], const unsigned char block[64]);  
void MD5Init(MD5_CTX *context);  
void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int  
inputLen);  
void MD5Final(unsigned char digest[16], MD5_CTX *context);  
void Encode(unsigned char *output, const uint32_t *input, unsigned int len);  
void Decode(uint32_t *output, const unsigned char *input, unsigned int len);  
void MD5_memcpy(unsigned char* output, const unsigned char* input, unsigned  
int len);  
void MD5_memset(unsigned char* output, int value, unsigned int len);
```

PADDING Array

```
unsigned char PADDING[64] = { 0x80, 0, 0, 0, ..., 0 };
```

This array is used to pad the input data to a multiple of 512 bits (64 bytes).

MD5Init Function

```
void MD5Init(MD5_CTX *context) {  
    context->count[0] = context->count[1] = 0;  
    context->state[0] = 0x67452301;  
    context->state[1] = 0xEFCDAB89;  
    context->state[2] = 0x98BADCFE;  
    context->state[3] = 0x10325476;  
}
```

Initializes the MD5 context. The ‘state’ array is initialized with a specific set of constants.

MD5Update Function

```
void MD5Update(MD5_CTX *context, const unsigned char *input, unsigned int inputLen)  
  
    unsigned int i, index, partLen;  
  
    index = (unsigned int)((context->count[0] >> 3) & 0x3F);
```

```

if ((context->count[0] += ((uint32_t)inputLen << 3)) < ((uint32_t)inputLen << 3))
    context->count[1]++;
context->count[1] += ((uint32_t)inputLen >> 29);

partLen = 64 - index;

if (inputLen >= partLen) {
    MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)input,
partLen);
    MD5Transform(context->state, context->buffer);

    for (i = partLen; i + 63 < inputLen; i += 64)
        MD5Transform(context->state, &input[i]);

    index = 0;
} else
    i = 0;

MD5_memcpy((unsigned char*)&context->buffer[index], (unsigned char*)&input[i],
inputLen-i);
}

```

Processes the input data in blocks of 512 bits (64 bytes). Updates the context with the new data and calls **MD5Transform** for each 512-bit block.

MD5Final Function

```
void MD5Final(unsigned char digest[16], MD5_CTX *context) {  
    unsigned char bits[8];  
    unsigned int index, padLen;  
  
    Encode(bits, context->count, 8);  
  
    index = (unsigned int)((context->count[0] >> 3) & 0x3f);  
    padLen = (index < 56) ? (56 - index) : (120 - index);  
    MD5Update(context, PADDING, padLen);  
  
    MD5Update(context, bits, 8);  
  
    Encode(digest, context->state, 16);  
  
    MD5_memset((unsigned char*)context, 0, sizeof (*context));  
}
```

Finalizes the MD5 hash computation. Pads the remaining data, appends the length of the input data, and computes the final hash value.

MD5Transform Function

```
void MD5Transform(uint32_t state[4], const unsigned char block[64]) {  
    uint32_t a = state[0], b = state[1], c = state[2], d = state[3], x[16];  
  
    Decode(x, block, 64);
```

```

#define S11 7

#define S12 12

#define S13 17

#define S14 22

#define S21 5

#define S22 9

#define S23 14

#define S24 20

#define S31 4

#define S32 11

#define S33 16

#define S34 23

#define S41 6

#define S42 10

#define S43 15

#define S44 21


#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))

#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))

#define H(x, y, z) ((x) ^ (y) ^ (z))

#define I(x, y, z) ((y) ^ ((x) | (~z)))


#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))


#define FF(a, b, c, d, x, s, ac) { \
    (a) += F ((b), (c), (d)) + (x) + (uint32_t)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}

```

```

#define GG(a, b, c, d, x, s, ac) { \
    (a) += G ((b), (c), (d)) + (x) + (uint32_t)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}

```

```

#define HH(a, b, c, d, x, s, ac) { \
    (a) += H ((b), (c), (d)) + (x) + (uint32_t)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}

```

```

#define II(a, b, c, d, x, s, ac) { \
    (a) += I ((b), (c), (d)) + (x) + (uint32_t)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
}

```

```

FF (a, b, c, d, x[ 0], S11, 0xd76aa478);
FF (d, a, b, c, x[ 1], S12, 0xe8c7b756);
FF (c, d, a, b, x[ 2], S13, 0x242070db);
FF (b, c, d, a, x[ 3], S14, 0xc1bdceee);
FF (a, b, c, d, x[ 4], S11, 0xf57c0faf);
FF (d, a, b, c, x[ 5], S12, 0x4787c62a);
FF (c, d, a, b, x[ 6], S13, 0xa8304613);
FF (b, c, d, a, x[ 7], S14, 0xfd469501);
FF (a, b, c, d, x[ 8], S11, 0x698098d8);
FF (d, a, b, c, x[ 9], S12, 0x8b44f7af);
FF (c, d, a, b, x[10], S13, 0xffff5bb1);
FF (b, c, d, a, x[11], S14, 0x895cd7be);
FF (a, b, c, d, x[12], S11, 0x6b901122);

```

```
FF (d, a, b, c, x[13], S12, 0xfd987193);
```

```
FF (c, d, a, b, x[14], S13, 0xa679438e);
```

```
FF (b, c, d, a, x[15], S14, 0x49b40821);
```

```
GG (a, b, c, d, x[ 1], S21, 0xf61e2562);
```

```
GG (d, a, b, c, x[ 6], S22, 0xc040
```

EXERCISE2

Create the 3 text files with different sentences and send these messages on the Kernel module and see that the MD5 device received these files?

SOLUTION:

with these command lines we created 3 test files 1 ,2,3 with different concepts.

```
echo "This is a test file." > testfile1.txt
```

```
echo "Another test file with different content." > testfile2.txt
```

```
echo "Hello, World!" > testfile3.txt
```

then again monitor kernel log:

```
sudo dmesg -w
```

then we should load the kernel module and verify it by:

```
sudo insmod md5_module.ko
```

```
lsmod | grep md5_module
```

then send the messages to the md5 device and see if it is received by the md5 device or not.

```
cat testfile1.txt | sudo tee /dev/md5_device
```

```
cat testfile2.txt | sudo tee /dev/md5_device
```

```
cat testfile3.txt | sudo tee /dev/md5_device
```

come back and see your md5 device. if your kernel is modified to seehash you can see that also.

CONCLUSION

In conclusion, it can be seen how the objectives set for the project were completed. The fundamental parts such as the algorithm and the driver were developed and then implemented in QEMU.