Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработать загружаемый модуль ядра, вычисляющий хеш очередей сообщений с использованием API ядра»

Студент **ИУ7-76Б** _____ **Ковель А.Д.**

Руководитель курсовой работы _____ **Рязанова Н.Ю.**

2023 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ-7
_____ И. В. Рудаков
«16» сентября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы

по теме

«Разработать загружаемый модуль ядра, вычисляющий хеш очередей сообщений с использованием API ядра»

Студент группы **ИУ7-76Б**

Ковель Александр Денисович

Направленность КР

учебная

Источник тематики

Курсовая работа кафедры

График выполнения КР: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание

Разработать загружаемый модуль ядра, вычисляющий хеш очередей сообщений с использованием API ядра.

Оформление научно-исследовательской работы:

Расчетно-пояснительная записка на **12-20** листах формата А4.

Дата выдачи задания «16» сентября 2023 г.

Руководитель курсовой работы	_____	**Рязанова Н.Ю**
	(Подпись, дата)	(Фамилия И. О.)
Студент	_____	**Ковель А. Д.**
	(Подпись, дата)	(Фамилия И. О.)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Аналитические раздел **5**

1.1 Прерывания 5

1.2 Обработчики аппаратных прерываний 5

1.3 Очереди работ 6

1.4 Хеширование 8

1.5 Шифрование 10

2 Конструкторская часть **15**

2.1 Разработка очереди работ 15

2.2 Разработка алгоритма обработчика шифрования 15

2.3 Разработка алгоритма обработчика хеширования 17

3 Технологическая часть **18**

3.1 Средства реализации 18

3.2 Структура курсового проекта 18

3.3 Листинг загружаемого модуля ядра 18

4 Исследовательская часть **28**

4.1 Технические характеристики 28

4.2 Демонстрация работы программы 28

ЗАКЛЮЧЕНИЕ **29**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **30**

ВВЕДЕНИЕ

Цель работы — разработка загружаемого модуль ядра, вычисляющий хеш очередей сообщений с использованием API ядра.

Для достижения поставленной цели, необходимо решить следующие задачи:

1) определить структуры, связанные с поставленной задачей;
2) проанализировать и выбрать алгоритм хеширования и шифрования, встроенного в ядро;
3) разработать алгоритм шифрования и хеширования очередей сообщений;
4) реализовать программное обеспечение;
5) провести исследование скорости работы очередей сообщений с шифрованием, хешированием и без.

1 Аналитические раздел

1.1 Прерывания

Прерывания делятся на:

1) исключения (деление на ноль, переполнение стека), данные тип прерывания является синхронным;
2) системные вызовы (программные) — вызываются с помощью команды из программы (int 21h), также являются синхронными;
3) аппаратные прерывания (прерывания от системного таймера, клавиатуры), эти же прерывания — асинхронные.

Прерывания разделяют на две группы: быстрые и медленные.

Для сокращения времени обработки медленных прерываний, они делятся на 2 части:

1) «top half» — верхняя половина, запускается в результате получения процессором сигнала прерывания;
2) «bottom half» — нижняя половина, отложенные вызовы.

Существуют несколько способов реализации «bottom half» обработчиков:

1) softirq;
2) tasklet (тасклеты);
3) workqueue (очереди работы).

1.2 Обработчики аппаратных прерываний

Обработчик аппаратного прерывания призван минимизировать объем необходимых действий и обеспечить как можно более быструю завершаемость. В типичном сценарии, указанный обработчик прерывания осуществляет сохранение полученных данных от внешнего устройства в ядерном буфере. Однако, с целью полноценной обработки прерываний, обработчик аппаратного прерывания должен инициировать помещение отложенного действия в очередь для его последующего выполнения.

Обработчики аппаратных прерываний представляют собой особые функ-

ции, которые вызываются операционной системой в ответ на возникновение прерывания от аппаратного устройства. Когда аппаратное в устройстве возникает прерывание (например, сигнализирует о завершении операции или возникновении ошибки), процессор прерывает текущее выполнение и передает управление на соответствующий обработчик прерывания.

Одной из основных задач обработчика аппаратного прерывания является сохранение состояния системы, осуществление необходимых операций для обработки прерывания и восстановление исходного состояния после завершения обработки. Обработчик может выполнять различные операции, такие как чтение данных из устройства, запись данных в память, обновление регистров и установка флагов. Кроме того, обработчик аппаратного прерывания может взаимодействовать с другими частями операционной системы, например, планировщиком задач, для оптимального распределения ресурсов и обработки прерываний в системе.

1.3 Очереди работ

Очереди работ представляют собой универсальный инструмент для отложенного выполнения операций, который позволяет функции обработчика блокироваться во время выполнения соответствующих действий. Очередь работ позволяет обработчику аппаратного прерывания добавлять необходимые операции для выполнения в очередь, вместо того чтобы выполнять их прямо внутри обработчика. Таким образом, функции обработчика могут блокироваться, ждать завершения определенных операций или условий, и продолжать работу только после их выполнения.

Листинг 1 – Структура workqueue_struct

```
struct workqueue_struct {
  struct list_head  pwqs;   /* WR: all pwqs of this wq */
  struct list_head  list;   /* PR: list of all workqueues */
  struct mutex    mutex;    /* protects this wq */
  int     work_color; /* WQ: current work color */
  int     flush_color;  /* WQ: current flush color */
```

```
  atomic_t    nr_pwqs_to_flush; /* flush in progress */
  struct wq_flusher *first_flusher; /* WQ: first flusher */
  struct list_head  flusher_queue;  /* WQ: flush waiters */
  struct list_head  flusher_overflow; /* WQ: flush overflow list */
  struct list_head  maydays;  /* MD: pwqs requesting rescue */
  struct worker   *rescuer; /* MD: rescue worker */

  int     nr_drainers;  /* WQ: drain in progress */
  int     saved_max_active; /* WQ: saved pwq max_active */

  struct workqueue_attrs  *unbound_attrs; /* PW: only for unbound wqs */
  struct pool_workqueue *dfl_pwq; /* PW: only for unbound wqs */

#ifdef CONFIG_SYSFS
  struct wq_device  *wq_dev;  /* I: for sysfs interface */
#endif
#ifdef CONFIG_LOCKDEP
  char      *lock_name;
  struct lock_class_key key;
  struct lockdep_map  lockdep_map;
#endif
  char      name[WQ_NAME_LEN]; /* I: workqueue name */

  /*
   * Destruction of workqueue_struct is RCU protected to allow walking
   * the workqueues list without grabbing wq_pool_mutex.
   * This is used to dump all workqueues from sysrq.
   */
  struct rcu_head   rcu;

  /* hot fields used during command issue, aligned to cacheline */
  unsigned int    flags ____cacheline_aligned; /* WQ: WQ_* flags */
  struct pool_workqueue __percpu *cpu_pwqs; /* I: per-cpu pwqs */
  struct pool_workqueue __rcu *numa_pwq_tbl[]; /* PWR: unbound pwqs indexed by
     node */
};
```

Листинг 2 – Структура work_struct

```
struct work_struct {
  atomic_long_t data;
```

```
  struct list_head entry;
  work_func_t func;
#ifdef CONFIG_LOCKDEP
  struct lockdep_map lockdep_map;
#endif
};
```

Работа может инициализироваться двумя способами:

- статически — DECLARE_WORK(name, void func), где: name — имя структуры work_struct, func — функция, которая вызывается из workqueue — обработчик «bottom half»;
- динамически — INIT_WORK(struct work_struct *work, void func).

После того, как будет инициализирована структура для объекта work, следующим шагом будет помещение этой структуры в очередь работ. Этом можно сделать несколькими способами. Во-первых, можно добавить работу (объект work) в очередь работ с помощью функции queue_work, которая назначает работу текущему процессу. Во-вторых, можно с помощью функции queue_work_on указать процессор, на котором будет выполняться обработчик.

1.4 Хеширование

Хеширование — это математический алгоритм, преобразовывающий произвольный массив данных в состоящую из букв и цифр строку фиксированной длины. Данный подход используется в нескольких подходах. Ускорение поиска в большом наборе данных или сокрытии информации. В данной работе рассматривается именно второй подход. Главным преимуществом хеширования является, то что нет необходимости передавать ключи для дешифрования сообщения, но также — это недостаток, так как любой человек, который узнает алгоритм, которым было произведено изменение текста, сможет получить исходный текст.

В ядре Linux предусмотрены специальные функции, отвечающие за инициализацию алгоритмов хеширования [1].

Листинг 3 – Функции алгоритмов хеширования.

```
struct crypto_shash *crypto_alloc_shash(const char *alg_name, u32 type, u32
    mask);
int crypto_shash_init(struct shash_desc *desc);
int crypto_shash_update(struct shash_desc *desc, const u8 *data, unsigned
    int len);
int crypto_shash_final(struct shash_desc *desc, u8 *out);
void crypto_free_shash(struct crypto_shash *tfm);
```

1) crypto_alloc_shash — выделяет память под обработчик хеширования. Возвращает структуру crypto_shash — это обработчик шифра, который требуется для любого последующего вызова API.
2) crypto_shash_init — инициализирует «сборник» сообщений, на который ссылается обработчик.
3) crypto_shash_update — обновляет состояние «сборника» сообщений в обработчике.
4) crypto_shash_final — завершает обработку «сборника» сообщений и создает выходное сообщение на основе всех данных обработчика.
5) crypto_free_shash — освобождение памяти обработчика хеширования.

В листинге 4 представлена структура обработчика хеширования [2].

Листинг 4 – Структура обработчика хеширования.

```
struct crypto_shash {
    unsigned int descsize;
    struct crypto_tfm base;
};
```

Структура состоит из descsize — размер рабочего состояния. Данный размер, необходим во время работы алгоритма хеширования и для расчета необходимой памяти, чтобы вызывающая сторона могла выделить достаточно памяти. crypto_tfm — является общей структурой для всех алгоритмов шифрования и хеширования. Создана для сокрытия и унификации функций всех алгоритмов.

В данной работе выбран алгоритм sha256 **??**, так как он является наиболее надежным на данный момент. Так как алгоритма хеширования являются не достаточно надежными с точки зрения сокрытия информации, необходимо рассмотреть функции шифрования.

1.5 Шифрование

Шифрование [3] — это технология кодирования и раскодирования данных. Зашифрованные данные — это результат применения алгоритма для кодирования данных с целью сделать их недоступными для чтения.

В ядре существуют специальные функции, отвечающие за регистрацию алгоритмов шифрования [4].

Листинг 5 – Регистрация алгоритмов шифрования.

```
/* include/linux/crypto.h */

int crypto_register_alg(struct crypto_alg *alg);
int crypto_register_algs(struct crypto_alg *algs, int count);

int crypto_unregister_alg(struct crypto_alg *alg);
int crypto_unregister_algs(struct crypto_alg *algs, int count);
```

Эти функции возвращают отрицательное значение в случае ошибки, и 0 — в случае успешного завершения, а регистрируемые алгоритмы описываются структурой crypto_alg.

Листинг 6 – Структура crypto_alg

```
/* include/linux/crypto.h */

struct crypto_alg {
    struct list_head cra_list;
    struct list_head cra_users;

    u32 cra_flags;
    unsigned int cra_blocksize;
    unsigned int cra_ctxsize;
```

```
    unsigned int cra_alignmask;

    int cra_priority;
    atomic_t cra_refcnt;

    char cra_name[CRYPTO_MAX_ALG_NAME];
    char cra_driver_name[CRYPTO_MAX_ALG_NAME];

    const struct crypto_type *cra_type;

    union {
        struct ablkcipher_alg ablkcipher;
        struct blkcipher_alg blkcipher;
        struct cipher_alg cipher;
        struct compress_alg compress;
    } cra_u;

    int (*cra_init)(struct crypto_tfm *tfm);
    void (*cra_exit)(struct crypto_tfm *tfm);
    void (*cra_destroy)(struct crypto_alg *alg);

    struct module *cra_module;
} CRYPTO_MINALIGN_ATTR;
```

- cra_flags: набор флагов, описывающих алгоритм.
- cra_blocksize: байтовый размер блока алгоритма. Все типы преобразований, кроме хэширования, возвращают ошибку при попытке обработать данные, размер которых меньше этого значения.
- cra_ctxsize: байтовый размер криптоконтекста. Ядро использует это значение при выделении памяти под контекст.
- cra_alignmask: маска выравнивания для входных и выходных данных. Буферы для входных и выходных данных алгоритма должны быть выровнены по этой маске.
- cra_priority: приоритет данной реализации алгоритма. Если в ядре зарегистрировано больше одного преобразования с одинаковым cra_name, то, при обращении по этому имени, будет возвращён алгоритм с наибольшим

приоритетом.

- cra_name: название алгоритма. Ядро использует это поле для поиска реализаций.
- cra_driver_name: уникальное имя реализации алгоритма.
- cra_type: тип криптопреобразования.
- cra_u: реализация алгоритма.
- cra_init: функция инициализации экземпляра преобразования. Эта функция вызывается единожды, во время создания экземпляра (сразу после выделения памяти под криптоконтекст).
- cra_exit: деинициализация экземпляра преобразования.

Но crypt_alg является неполной структурой и сейчас используется усовершенствованный вариант struct skcipher_alg. Ее также необходимо зарегистрировать как и crypto_alg.

Листинг 7 – Структура skcipher_alg

```
    /* include/crypto/skcipher.h */

struct skcipher_alg {
    int (*setkey)(struct crypto_skcipher *tfm, const u8 *key,
                  unsigned int keylen);
    int (*encrypt)(struct skcipher_request *req);
    int (*decrypt)(struct skcipher_request *req);
    int (*init)(struct crypto_skcipher *tfm);
    void (*exit)(struct crypto_skcipher *tfm);

    unsigned int min_keysize;
    unsigned int max_keysize;
    unsigned int ivsize;
    unsigned int chunksize;
    unsigned int walksize;

    struct crypto_alg base;
};
```

В этой структуре используется crypto_alg, описывающий алгоритм. Из

важных полей тут:

- chunksize: данное поле отвечает за размер блока шифрования (если этот блок не относится к поточным).
- walksize: равен значению chunksize, за исключением случаев, когда алгоритм может параллельно обрабатывать несколько блоков, тогда walksize может быть больше, чем chunksize, но обязательно должен быть кратен ему.

Также в в функциях encrypt и decrypt присутствует структура skcipher_request. Данная структура содержит данные, необходимые для выполнения операции симметричного шифрования.

В Crypto API есть еще некоторые особенности. Например, все алгоритмы шифрования данных произвольной длины работают со входными данными не через указатели на байтовые массивы, а через структуру scatterlist.

Листинг 8 – Структура skcipher_alg

```
/* include/linux/scatterlist.h */

struct scatterlist {
    /* ... */
    unsigned long   page_link;
    unsigned int    offset;
    unsigned int    length;
    /* ... */
};
```

Экземпляр этой структуры можно проинициализировать указателем на некоторые данные. Например, при помощи вызова функции sg_init_one. В этой функции определяется страница памяти, с которой «начинается» buf (page_link), и определяется смещение указателя buf относительно адреса начала страницы (offset). Таким образом, криптографическая подсистема работает напрямую со страницами памяти.

В данной работе представлено шифрование на примере алгоритма AES [5].

Вывод

В данном разделе были представлены алгоритмы: очередей работ, хеширования, шифрования. Также было показано, что хеширование является менее надежным алгоритмом, чем шифрование. Для хеширования был выбран алгоритм sha256, а для шифрования AES.

2 Конструкторская часть

2.1 Разработка очереди работ

На рисунке 1 представлена схема алгоритма очереди работ.

Рис. 1 – Алгоритм очереди работ

2.2 Разработка алгоритма обработчика шифрования

После того как очереди работ считают клавишу с клавиатуры, начинает работать алгоритм шифрования. На рисунке 3 представлена схема алгоритма шифрования.

Рис. 2 – Алгоритм шифрования

2.3 Разработка алгоритма обработчика хеширования

Также для сравнения представлена схема алгоритма хеширования.

Рис. 3 – Алгоритм шифрования

3 Технологическая часть

3.1 Средства реализации

Для реализации ПО был выбран язык C [6]. В данном языке есть все требующиеся инструменты для данной курсовой работы. В качестве среды разработки была выбрана среда VS code [7].

3.2 Структура курсового проекта

Курсовой проект состоит из:

- my_ascii.h — файл содержащий код клавиш, которые будут анализироваться очередью работ;
- workqueue_struct.h — содержит структуру очереди работ;
- cryptosk.c — загружаемый модуль ядра, содержащий основную логику программы;
- hash.c — функция хеширования;
- hash.h — файл для получение доступа к функция хеширования.

3.3 Листинг загружаемого модуля ядра

На листинге 10 приведен код загружаемого моду ядра.

Листинг 9 – Загружаемый модуль ядра

```
/*
* cryptosk.c
*/
#include <crypto/internal/skcipher.h>
#include <linux/crypto.h>
#include <linux/module.h>
#include <linux/random.h>
#include <linux/scatterlist.h>

#include <linux/kernel.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <asm/io.h>
#include <linux/stddef.h>
#include <linux/workqueue.h>
```

```
#include <linux/delay.h>


#include "my_ascii.h"


#define SYMMETRIC_KEY_LENGTH 32
#define CIPHER_BLOCK_SIZE 16

struct tcrypt_result {
    struct completion completion;
    int err;
};

struct skcipher_def {
    struct scatterlist sg;
    struct crypto_skcipher *tfm;
    struct skcipher_request *req;
    struct tcrypt_result result;
    char *scratchpad;
    char *ciphertext;
    char *ivdata;
};

static struct skcipher_def sk;

typedef struct
{
    struct work_struct work;
    int code;
} my_work_struct_t;

static struct workqueue_struct *my_wq;

static my_work_struct_t *work1;

int keyboard_irq = 1;
char *password = "password123";
```

```
static void test_skcipher_finish(struct skcipher_def *sk)
{
    if (sk->tfm)
        crypto_free_skcipher(sk->tfm);
    if (sk->req)
        skcipher_request_free(sk->req);
    if (sk->ivdata)
        kfree(sk->ivdata);
    if (sk->scratchpad)
        kfree(sk->scratchpad);
    if (sk->ciphertext)
        kfree(sk->ciphertext);
}

static int test_skcipher_result(struct skcipher_def *sk, int rc)
{
    switch (rc) {
    case 0:
        break;
    case -EINPROGRESS || -EBUSY:
        rc = wait_for_completion_interruptible(&sk->result.completion);
        if (!rc && !sk->result.err) {
            reinit_completion(&sk->result.completion);
            break;
        }
        pr_info("skcipher encrypt returned with %d result %d\n", rc,
                sk->result.err);
        break;
    default:
        pr_info("skcipher encrypt returned with %d result %d\n", rc,
                sk->result.err);
        break;
    }

    init_completion(&sk->result.completion);

    return rc;
}
```

```
static void test_skcipher_callback(void *req, int error)
{
    struct crypto_async_request *res = req;
    struct tcrypt_result *result = res->data;

    if (error == -EINPROGRESS) {
        pr_info("Error EINPROGRESS\n");
        return;

    }

    result->err = error;
    complete(&result->completion);
    pr_info("Encryption finished successfully\n");

    /*          . */

    memset((void*)sk.scratchpad, '-', CIPHER_BLOCK_SIZE);
    int ret = crypto_skcipher_decrypt(sk.req);
    ret = test_skcipher_result(&sk, ret);
    if (ret) {
        pr_info("Error test_skcipher_result\n");
        return;

    }

    sg_copy_from_buffer(&sk.sg, 1, sk.scratchpad, CIPHER_BLOCK_SIZE);
    sk.scratchpad[CIPHER_BLOCK_SIZE-1] = 0;

    pr_info("Decryption request successful\n");
    pr_info("Decrypted: %s\n", sk.scratchpad);

}

static int test_skcipher_encrypt(char *plaintext, char *password,
                                 struct skcipher_def *sk)
{
    int ret = -EFAULT;
    unsigned char key[SYMMETRIC_KEY_LENGTH];
```

```
if (!sk->tfm) {
    sk->tfm = crypto_alloc_skcipher("cbc-aes-aesni", 0, 0);
    if (IS_ERR(sk->tfm)) {
        pr_info("could not allocate skcipher handle\n");
        return PTR_ERR(sk->tfm);
    }
}

if (!sk->req) {
    sk->req = skcipher_request_alloc(sk->tfm, GFP_KERNEL);
    if (!sk->req) {
        pr_info("could not allocate skcipher request\n");
        ret = -ENOMEM;
        return ret;
    }
}

skcipher_request_set_callback(sk->req, CRYPTO_TFM_REQ_MAY_BACKLOG,
                              test_skcipher_callback, &sk->result);

/*        . */
memset((void *)key, '\0', SYMMETRIC_KEY_LENGTH);

sprintf((char *)key, "%s", password);

if (crypto_skcipher_setkey(sk->tfm, key, SYMMETRIC_KEY_LENGTH)) {
    pr_info("key could not be set\n");
    ret = -EAGAIN;
    return ret;
}
pr_info("Symmetric key: %s\n", key);
pr_info("Plaintext: %s\n", plaintext);

if (!sk->ivdata) {
    sk->ivdata = kmalloc(CIPHER_BLOCK_SIZE, GFP_KERNEL);
    if (!sk->ivdata) {
        pr_info("could not allocate ivdata\n");
        return ret;
    }
    get_random_bytes(sk->ivdata, CIPHER_BLOCK_SIZE);
```

```
    }

    if (!sk->scratchpad) {
        /*            . */
        sk->scratchpad = kmalloc(CIPHER_BLOCK_SIZE, GFP_KERNEL);
        if (!sk->scratchpad) {
            pr_info("could not allocate scratchpad\n");
            return ret;
        }
    }
    sprintf((char *)sk->scratchpad, "%s", plaintext);

    sg_init_one(&sk->sg, sk->scratchpad, CIPHER_BLOCK_SIZE);
    skcipher_request_set_crypt(sk->req, &sk->sg, &sk->sg, CIPHER_BLOCK_SIZE,
                               sk->ivdata);
    init_completion(&sk->result.completion);

    /*          . */
    ret = crypto_skcipher_encrypt(sk->req);
    ret = test_skcipher_result(sk, ret);
    if (ret)
        return ret;

    pr_info("Encrypted texted: %s\n", (char *)sk->scratchpad);
    pr_info("Encryption request successful\n");


    return ret;
}

void work1_func(struct work_struct *work)
{
    my_work_struct_t *my_work = (my_work_struct_t *)work;
    int code = my_work->code;

    printk(KERN_INFO "MyWorkQueue: work1 begin");

    if (code < 84)
        printk(KERN_INFO "MyWorkQueue: the key is %s", ascii[code]);
```

```
    printk(KERN_INFO "MyWorkQueue: work1 end");
}

irqreturn_t my_irq_handler(int irq, void *dev)
{
    int code;
    printk(KERN_INFO "MyWorkQueue: my_irq_handler");

    if (irq == keyboard_irq)
    {
        printk(KERN_INFO "MyWorkQueue: called by keyboard_irq");

        code = inb(0x60);
        work1->code = code;

        unsigned char mesage[SYMMETRIC_KEY_LENGTH];

        sprintf((char *)mesage, "%d", code);

        test_skcipher_encrypt((char *)mesage, password, &sk);


        queue_work(my_wq, (struct work_struct *)work1);

        return IRQ_HANDLED;
    }

    printk(KERN_INFO "MyWorkQueue: called not by keyboard_irq");

    return IRQ_NONE;
}

static int __init my_workqueue_init(void)
{
    int ret;

    sk.tfm = NULL;
    sk.req = NULL;
    sk.scratchpad = NULL;
    sk.ciphertext = NULL;
```

```
    sk.ivdata = NULL;

    ret = request_irq(keyboard_irq, my_irq_handler, IRQF_SHARED,
                      "test_my_irq_handler", (void *) my_irq_handler);

    printk(KERN_INFO "MyWorkQueue: init");
    if (ret)
    {
        printk(KERN_ERR "MyWorkQueue: request_irq error");
        return ret;
    }
    else
    {
        my_wq = alloc_workqueue("%s", __WQ_LEGACY | WQ_MEM_RECLAIM, 1,
           "my_wq");

        if (my_wq == NULL)
        {
            printk(KERN_ERR "MyWorkQueue: create queue error");
            ret = GFP_NOIO;
            return ret;
        }

        work1 = kmalloc(sizeof(my_work_struct_t), GFP_KERNEL);
        if (work1 == NULL)
        {
            printk(KERN_ERR "MyWorkQueue: work1 alloc error");
            destroy_workqueue(my_wq);
            ret = GFP_NOIO;
            return ret;
        }

        INIT_WORK((struct work_struct *)work1, work1_func);
        printk(KERN_ERR "MyWorkQueue: loaded");
    }
    return ret;
}

static void __exit my_workqueue_exit(void)
{
```

```
        printk(KERN_INFO "MyWorkQueue: exit");

        synchronize_irq(keyboard_irq); //
        free_irq(keyboard_irq, my_irq_handler); //

        flush_workqueue(my_wq);
        destroy_workqueue(my_wq);
        kfree(work1);



        printk(KERN_INFO "MyWorkQueue: unloaded");

        test_skcipher_finish(&sk);

        printk(KERN_INFO "Crypto: unloaded");
    }

    module_init(my_workqueue_init);
    module_exit(my_workqueue_exit);

    MODULE_DESCRIPTION("Symmetric key encryption example");
    MODULE_LICENSE("GPL");
    MODULE_AUTHOR("Kovel A.");
```

Листинг 10 – Функия хеширования

```
static void show_hash_result(char *plaintext, char *hash_sha256)
{
    int i;
    char str[SHA256_LENGTH * 2 + 1];

    pr_info("sha256 test for string: \"%s\"\n", plaintext);
    for (i = 0; i < SHA256_LENGTH; i++)
        sprintf(&str[i * 2], "%02x", (unsigned char)hash_sha256[i]);
    str[i * 2] = 0;
    pr_info("%s\n", str);
}

static int cryptosha256_init(char *plaintext)
{
```

```
    char hash_sha256[SHA256_LENGTH];
    struct crypto_shash *sha256;
    struct shash_desc *shash;

    sha256 = crypto_alloc_shash("sha256", 0, 0);
    if (IS_ERR(sha256))
        return -1;

    shash = kmalloc(sizeof(struct shash_desc) + crypto_shash_descsize(sha256),
                    GFP_KERNEL);
    if (!shash)
        return -ENOMEM;

    shash->tfm = sha256;

    if (crypto_shash_init(shash))
        return -1;

    if (crypto_shash_update(shash, plaintext, strlen(plaintext)))
        return -1;

    if (crypto_shash_final(shash, hash_sha256))
        return -1;

    kfree(shash);
    crypto_free_shash(sha256);

    show_hash_result(plaintext, hash_sha256);

    return 0;
}
```

4 Исследовательская часть

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [8] Linux [9];
- Оперативная память 32 Гбайт;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [10].

4.2 Демонстрация работы программы

Замеры времени проводились с выполнением алгоритма 100 раз.

На рисунке 5 представлен результат работы шифрования и время выполнения (644209 мкс.).

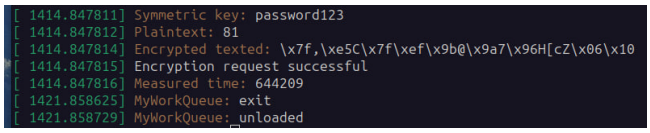

Рис. 4 – Результат работы с шифрованием

На рисунке 5 представлен результат работы хеширования и время выполнения (490961 мкс.).

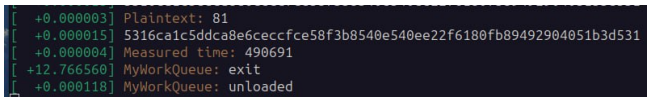

Рис. 5 – Результат работы с хешированием

На рисунке 6 представлен результат работы без шифрования и время выполнения (329342 мкс.).

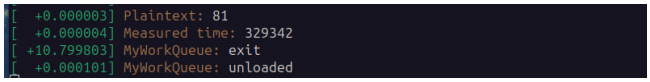

Рис. 6 – Результат работы без шифрования

Из данных результатов следует, что алгоритм с шифрованием работает в 2 раз медленнее, чем без него, а хеширование в 1.5 раза медленнее.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были выполнены следующие задачи:

1) определены структуры, связанные с поставленной задачей;
2) проанализированы и выбраны алгоритм хеширования и шифрования, встроенного в ядро;
3) разработан алгоритм шифрования и хеширования очередей сообщений;
4) реализованно программное обеспечение;
5) проведено исследование скорости работы очередей сообщений с шифрованием, хешированием и без.

Было выявлено, что алгоритм шифрования работает почти в полтора раза медленнее, чем хеширование и также требует ввода ключа для получения строки обратно. Из этого следует, что если необходима скорость сокрытия информации, стоит использовать хеширование. Для более надежного хранения стоит использовать шифрование.

Поставленная цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Message Digest Algorithm Definitions [Электронный ресурс]. Режим доступа: https://docs.kernel.org/crypto/api-digest.html (дата обращения: 19.12.2022).

2. Структура обработчика хеширования [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/crypto/hash.h#L268 (дата обращения: 19.12.2022).

3. Что такое шифрование? [Электронный ресурс]. Режим доступа: https://www.kaspersky.ru/resource-center/definitions/encryption (дата обращения: 19.12.2022).

4. Developing Cipher Algorithms [Электронный ресурс]. Режим доступа: https://docs.kernel.org/crypto/devel-algos.html (дата обращения: 19.12.2022).

5. AES [Электронный ресурс]. Режим доступа: https://blog.kraden.com/ru/aes-256-encryption (дата обращения: 19.12.2022).

6. Язык программирования C [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/cpp/c-language/?view=msvc-170. дата обращения: 12.11.2023.

7. Vscode [Электронный ресурс]. https://code.visualstudio.com/. дата обращения: 12.11.2023.

8. Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: https://pop.system76.com (дата обращения: 20.11.2022).

9. Linux – Документация [Электронный ресурс]. Режим доступа: https://docs.kernel.org (дата обращения: 20.11.2022).

10. Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [Электронный ресурс]. Режим доступа: https://www.amd.com/en/products/cpu/amd-ryzen-7-2700 (дата обращения: 20.11.2022).