**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 «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
# *К КУРСОВОЙ РАБОТЕ*
# *НА ТЕМУ:*
# *Исследование распределения памяти в многопоточных приложениях Linux*

Студент ИУ7-74Б _____ Новиков М. Р.
(Группа) (Подпись, дата) (Фамилия И. О.)

Руководитель курсовой работы _____ Рязанова Н. Ю.
(Подпись, дата) (Фамилия И. О.)

*2020 г.*

# Содержание

# Введение

В настоящее время с ростом объемов приложений — кода и объема обрабатываемых данных — важнейшей задачей является мониторинг потребления процессом ресурсов системы, в частности памяти. Так, например, распределение памяти в многопоточных приложениях Linux может отличаться от распределения памяти в однопоточных аналогах тех же программ. Распределение памяти в многопоточных приложениях может приводить неэффективному использованию памяти и, как следствие, к замедлению работы программы и системы в целом. В связи с этим, помимо стандартного средства выделения памяти в Linux (функции `malloc()` низкоуровневой библиотеки glibc), разработчики предлагают использование таких менеджеров памяти, как mimalloc от корпорации Microsoft, tcmalloc от Google, jemalloc, разработанный Джейсоном Эвансом для FreeBSD 7.0.

Курсовая работа посвящена разработке программного обеспечения для перехвата системных вызовов с целью анализа информации об адресном пространстве процесса и исследованию распределения памяти в многопоточных приложения Linux.

# 1 Аналитический раздел

## 1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу, необходимо исследовать распределение памяти в многопоточных приложениях Linux.

Для решения поставленной задачи необходимо:

1. Провести анализ стандартного аллокатора памяти.
2. Провести анализ системных вызовов выделения памяти.
3. Провести анализ и выбрать метод перехвата системных вызовов.
4. Разработать алгоритмы перехвата системных вызовов и вывода данных о потреблении процессом памяти.
5. Разработать загружаемый модуль ядра, реализующий поставленную задачу.
6. Разработать тестовые однопоточные и многопоточные программы для исследования выделения памяти.
7. Исследовать выделение памяти для многопоточных приложений.

## 1.2 Анализ стандартных функций распределения памяти

Функция `malloc()` библиотеки glibc выделяет память из кучи и при необходимости регулирует размер кучи с помощью функции `sbrk()`,

которая является оберткой системного вызова `brk`. Куча является виртуальной, то есть располагается в виртуальном адресном пространстве процесса. При выделении блоков памяти размером больше `MMAP_THRESHOLD` байтов память выделяется как анонимное отображение с помощью системного вызова `mmap`. `MMAP_THRESHOLD` по умолчанию равен 128 килобайтам, но может быть изменен с помощью `mallopt()`.

Память освобождается путем вызова функции `free()`, которая выполняет системный вызов `brk` или `munmap`.

## 1.3 Анализ системных вызовов brk, mmap и munmap

Размер сегмента данных адресного пространства процесса управляется системными вызовами `brk`, `mmap` и `munmap` (см. листинг 1.1).

Листинг 1.1: Прототипы системных вызовов `brk`, `mmap` и `munmap`

```
1 #include <unistd.h>
2 #include <sys/mman.h>
3
4 int brk(void *addr);
5 void *mmap(void *addr, size_t length, int prot, int flags, int fd,
     off_t offset);
6 int munmap(void *addr, size_t length);
```

Вызов `brk` устанавливает разрыв программы (program break) в значение, указанное в аргументе `addr`. Разрыв программы — это адрес первого байта, расположенного после сегмента данных процесса. Таким образом, увеличение разрыва программы позволяет выделить память в куче, а уменьшение — освободить. Функция `sbrk()` является оберткой функции `brk`, не считается системным вызовом и позволяет увеличивать сегмент данных на указанное в аргументе количество байтов.

Вызов `mmap` отображает `length` байтов, начиная со смещения `offset` в файле, определенного файловым дескриптором `fd`, в память, начиная с адреса `addr`. Возвращает `mmap` указатель на отображенные данные. Начиная с ядра версии 2.4 поддерживается флаг `MAP_ANONYMOUS`, позволяющий реализовывать анонимные отображения. Анонимное отображение — это отображение пространства виртуальной памяти процесса, а не файла в

пространстве файловой системы. Анонимные отображения не являются частью стандарта POSIX, однако реализованы во многих системах.

Вызов `munmap` удаляет отображение в память по адресу `addr` размера `lenght`.

Адресное пространство процесса описывается в ядре с помощью структуры `mm_struct`, которая называется дескриптором памяти и определена в `<linux/mm_types.h>`. Некоторые поля структуры `mm_struct` представлены на листинге 1.2.

Листинг 1.2: Структура `mm_struct`

```
struct mm_struct {
        struct vm_area_struct *mmap; /* Список областей памяти */
        struct rb_root mm_rb;
        u64 vmacache_seqnum;
        ...
        unsigned long mmap_base; /* Начальный адрес mmap */
        ...
        unsigned long task_size; /* Размер виртуального адресного
           пространства */
        ...
        int map_count; /* Количество областей памяти */
        ...
        unsigned long hiwater_rss; /* Наибольшее потребление RSS */
        unsigned long hiwater_vm; /* Наибольшее потребление виртуальной
           памяти */

        unsigned long total_vm; /* Общее количество страниц памяти */
        unsigned long locked_vm; /* Количество заблокированных страниц
           памяти */
        ...

        spinlock_t arg_lock; /* Спин-блокировки для нижерасположенных
           переменных */
        unsigned long start_code, end_code; /* Начало и конец области
           кода */
        unsigned long start_data, end_data; /* Начало и конец области
           данных */
        unsigned long start_brk, brk; /* Начало и конец кучи */
        unsigned long start_stack; /* Начало стека */

        ...
        struct mm_rss_stat rss_stat; /* Счетчики RSS */
        ...
};
```

Структура `vm_area_struct` описывает область памяти процесса и

является двусвязным списком. На листинге 1.3 представлены некоторые поля этой структуры.

Листинг 1.3: Структура `vm_area_struct`

```
struct vm_area_struct {
        unsigned long vm_start; /* Начало области памяти */
        unsigned long vm_end; /* Конец области памяти */

        struct vm_area_struct *vm_next, *vm_prev; /* Следующая и
           предыдущая область памяти */
        ...
        struct mm_struct *vm_mm; /* Адресное пространство, к которому
           принадлежит области памяти */
        ...
        unsigned long vm_flags;
        ...
        struct anon_vma *anon_vma; /* Описание анонимного отображения */
        ...
        struct file *vm_file; /* Отображенный файл */
        ...
}
```

Размер области памяти можно определить как разность `vm_end` и `vm_start`. При этом если `vma->vm_start <= mm->brk && vma->vm_end >= mm->start_brk`, где `vma` — рассматриваемая области памяти, `mm` — рассматриваемое адресное пространство, то данная область памяти является кучей. Если `vma->vm_start <= mm->start_stack && vma->vm_end >= mm->start_stack`, то данная область памяти является стеком. Если `vma->anon_vma != NULL && vma->vm_file == NULL`, то область памяти является анонимным отображением.

## 1.4 Методы перехвата системных вызовов

### 1.4.1 Сплайсинг

Сплайсинг — метод перехвата системных вызовов путем добавления инструкций безусловного перехода для встраивания собственного кода в оригинальный системный вызов. Реализуется метод на языке ассемблера.

Из преимуществ данного метода можно отметить отсутствие специальных требований к ядру и минимальные накладные расходы на перехват. Главный недостаток — высокая сложность реализации.

### 1.4.2 Модификация таблицы системных вызовов

Данный метод заключается в замене адреса перехватываемого системного вызова в таблице системных вызовов на адрес некоторой функции-перехватчика, которая кроме собственно системного вызова выполняет ряд других необходимых в рамках поставленной задачи операций (например, журналирование).

Для перехвата необходимо экспортировать таблицу системных вызовов, определить указатель для сохранения оригинального вывоза, реализовать собственный системный вызов, во время инициализации загружаемого модуля сохранить указатель на оригинальный вызов, заменить адрес вызова в системной таблице. При выгрузке модуля необходимо восстановить адрес оригинального вызова. На листинге 1.4 представлен пример перехвата вызова `mkdir` — реализованная функция не выполняет оригинальный вызов и возвращает нуль.

Листинг 1.4: Перехват системного вызова `mkdir` (ядро версии 2.4.22)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <sys/syscall.h>

extern void *sys_call_table[];
int (*orig_mkdir)(const char *path);

int own_mkdir(const char *path)
{
        return 0;
}

int init_module(void)
{
        orig_mkdir = sys_call_table[SYS_mkdir];
        sys_call_table[SYS_mkdir] = own_mkdir;
        return(0);
}

void cleanup_module(void)
{
        sys_call_table[SYS_mkdir] = orig_mkdir;
}
```

Начиная с ядра версии 2.5.41 символ `sys_call_table` более не экспортируется, так как возможность замены адресов вызовов в системной

таблице является потенциально опасной. Для определения адреса таблицы системных вызовов в ядрах 2.6 и новее следует рассмотреть файл `/proc/kallsyms`, содержащий адреса всех символов ядра. Получить адрес искомой таблицы можно выполнив команду `# cat /proc/kallsyms | grep " sys_call_table"`. Адрес таблицы позволит определить адреса оригинальных системных вызовов и выполнить необходимые замены.

### 1.4.3 Linux Security Modules

Linux Security Modules — фреймворк для разработки модулей безопасности ядра Linux. Начиная с ядра версии 2.6 LSM является частью ядра. На основе LSM разработаны AppArmor, SELinux и некоторые другие модули безопасности.

Для некоторых системных вызовов фреймворк реализует функции-перехватчики, в которые возможно встраивать собственный код. Это позволяет работать с системными вызовами без необходимости модификации системной таблицы.

Для перехвата `mkdir` следует в директории `/security` создать поддиректорию для модуля и исходный код модуля (см. листинг 1.5). Макрос `LSM_HOOK_INIT` регистрирует соответствие `foobar_inode_mkdir()` функции-перехватчику `inode_mkdir()`, а `security_add_hooks()` добавляет функцию в общий список пользовательских функций-перехватчиков LSM. Далее необходимо добавить заголовок пользовательской функции-перехватчика в файл `/include/linux/lsm_hooks.h` (см. листинг 1.6).

Листинг 1.5: Функция-перехватчик `mkdir`

```
1  #include <linux/module.h>
2  #include <linux/lsm_hooks.h>
3
4  static int foobar_inode_mkdir(struct inode *dir, struct dentry *dentry,
       umode_t mask)
5  {
6          printk(KERN_INFO "mkdir");
7          return 0;
8  }
9
10 static struct security_hook_list foobar_hooks[] =
11 {
12         LSM_HOOK_INIT(inode_mkdir, foobar_inode_mkdir),
```

```
};

void __init foobar_add_hooks(void)
{
        security_add_hooks(foobar_hooks, ARRAY_SIZE(foobar_hooks));
}
```

Листинг 1.6: Регистрация функции-перехватчика `mkdir`

```
#ifdef CONFIG_SECURITY_FOOBAR
        extern void __init foobar_add_hooks(void);
#else
        static inline void __init foobar_add_hooks(void) { }
#endif
```

Достоинство данного метода заключается в возможности более безопасной работы с кодом ядра. Среди недостатков данного метода: необходимость пересборки ядра, ограничение в один модуль безопасности. Главной задачей фреймворка LSM остается реализация дополнительных проверок прав доступа.

## 1.5 Вывод

В результате анализа особенностей работы стандартного аллокатора памяти было установлено, что аллокатор для выделения памяти использует системные вызовы `brk` и `mmap`.

В результате анализа системных вызовов были определены их назначения, особенности реализации, особенности описания адресного пространства процесса в ядре операционной системы Linux.

В качестве метода для перехвата системных вызовов был выбран метод модификации системной таблицы, как наиболее гибкий в условиях поставленной задачи и не требующий реализации на языке ассемблера.

# 2 Конструкторский раздел

## 2.1 Проектирование загружаемого модуля ядра

При загрузке модуль ядра получает адрес таблицы системных вызовов в шестнадцатеричном виде, конвертирует его в указатель, инициализирует функции-перехватчики для системных вызовов `brk`, `mmap` и `munmap`, модифицируя системную таблицу.

При вызове одной из перехваченных функций выполняется функция журналирования данных о размере областей памяти адресного пространства вызывающего процесс. В системный журнал записывается информация о размерах кучи, стека и анонимных отображений.

При выгрузке модуля выполняется обратная замена адресов в таблице системных вызовов и освобождение памяти, выделенной под хранение необходимых структур данных.

На рисунке 2.1 представлена структура программного обеспечения.

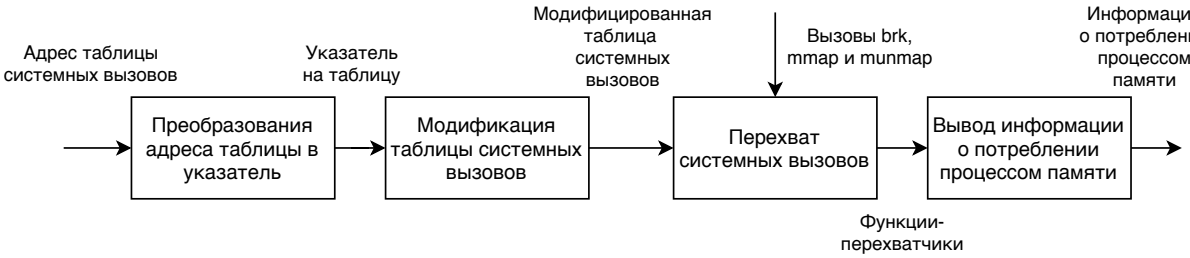


Рис. 2.1: Структура загружаемого модуля ядра

### 2.1.1 Алгоритмы перехвата системных вызовов

Алгоритм встраивания функций-перехватчиков представлен на рисунке 2.2.

Вызов подпрограммы `sys_hook_init()` инициализирует структуру данных, описывающую функции-перехватчики. Далее инициализируются структуры данных, описывающие каждый конкретный перехватчик.

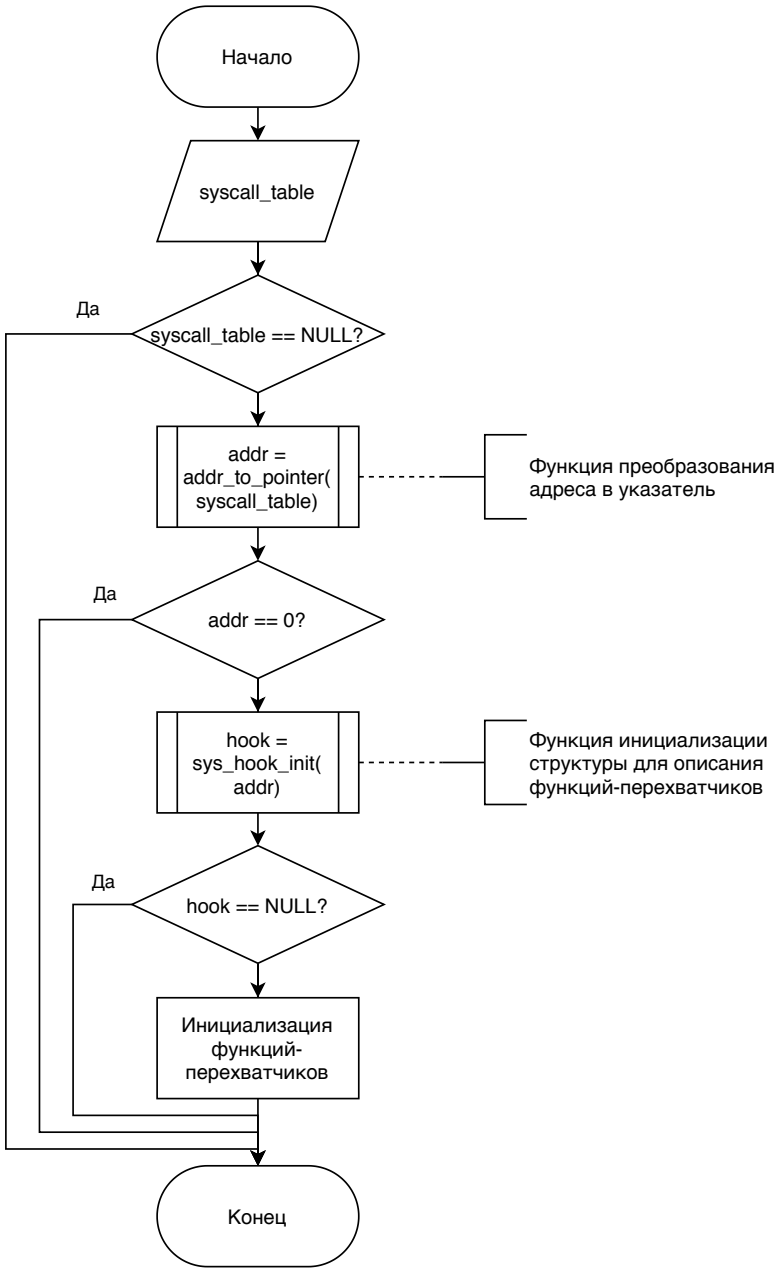


Рис. 2.2: Алгоритм встраивания функций-перехватчиков

Все функции-перехватчики одинаковы по своей структуре: определяется адрес оригинального системного вызова, выполняется оригинальный вызов, затем вызывается функция журналирования данных. Схема алгоритма функции-перехватчика представлена на рисунке 2.3.


Рис. 2.3: Алгоритм функции-перехватчика

### 2.1.2 Алгоритм журналирования данных ядра

Схема алгоритма журналирования размера областей памяти адресного пространства процесса представлена на рисунке 2.4.

## 2.2 Вывод

В результате проектирования разработаны алгоритмы перехвата системных вызовов и вывода данных о потреблении процессом памяти, что позволяет перейти к реализации загружаемого модуля в программном коде.

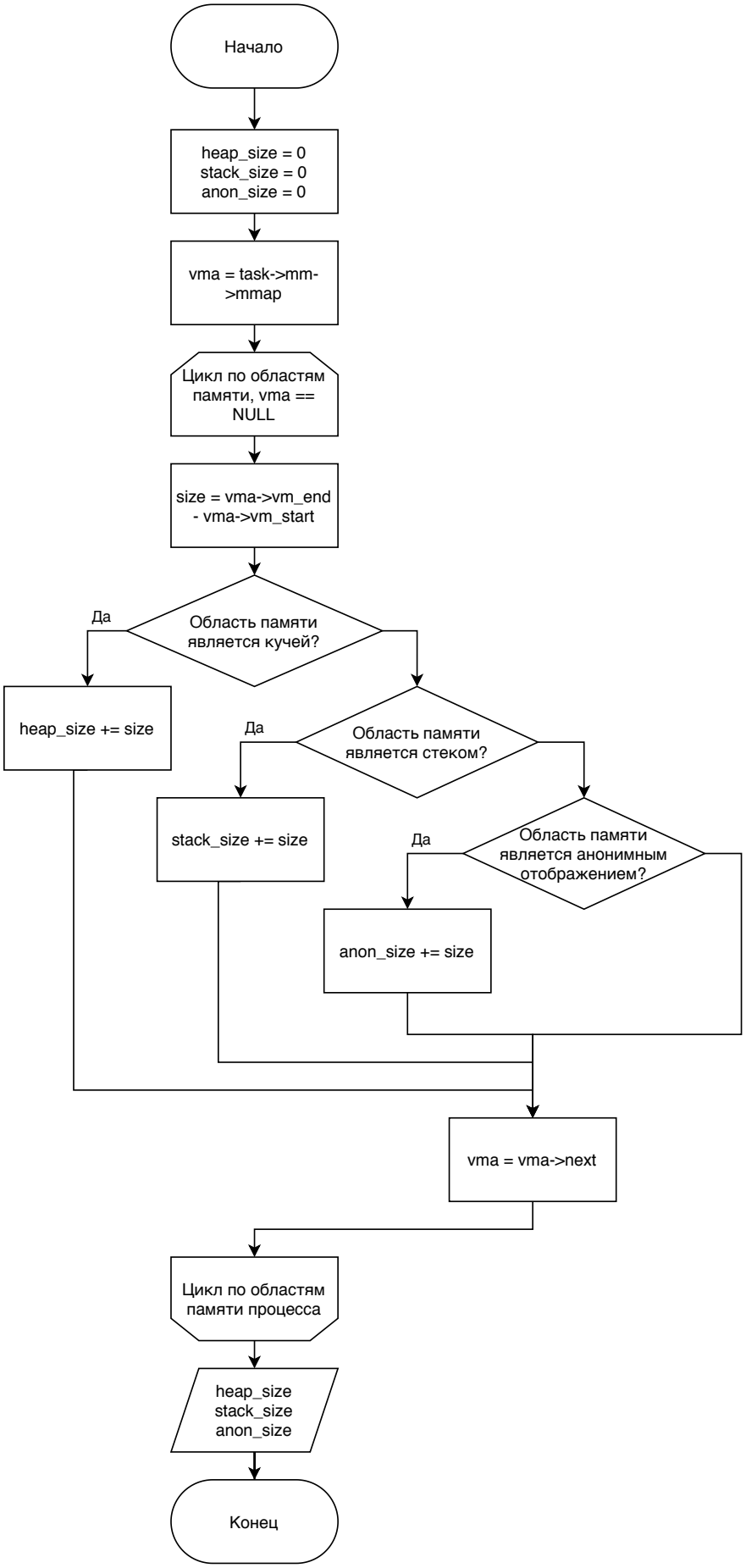


Рис. 2.4: Алгоритм функции журналирования

# 3 Технологический раздел

## 3.1 Выбор языка программирования и среды разработки

В качестве языка программирования был выбран язык C, как наиболее предпочтительный при разработке загружаемых модулей ядра Linux. Для сборки загружаемого модуля была выбрана утилита make.

В качестве среды разработки была выбрана программа CLion, разрабатываемая компанией JetBrains. CLion содержит редактор кода, отладчик, средства для статического анализа кода, средства для сборки проекта.

## 3.2 Описание структур данных

Для описания функций-перехватчиков в загружаемом модуле требуется структура данных, содержащая указатель на таблицу системных вызовов, указатели на начало и конец списка функций-перехватчиков.

Каждая функция-перехватчик описывается структурой (списком), содержащей идентификатор системного вызова, указатель на оригинальный системный вызов, указатель на функцию-перехватчик, указатель на следующую структуру. В условиях поставленной задачи можно ограничиться односвязным списком. При инициализации новой функции следует добавить в конец списка новый элемент, выделив под него память.

## 3.3 Реализация загружаемого модуля ядра

На листинге 4.3 Приложения А представлен Makefile загружаемого модуля ядра. Кроме непосредственно сборки модуля Makefile содержит цели `install`, `uninstall` и `clean`, которые позволяют загрузить, выгрузить модуль и очистить директорию от файлов сборки.

Для загрузки модуля необходимо определить адрес таблицы системных вызовов. В данном случае используется команда `# cat /proc/kallsyms | grep " sys_call_table"`. Затем извлекается первое полученное в результате выполнения команды значение. Полученный адрес передается в модуль в качестве параметра `syscall_table_addr` при исполнении команды `# insmod ./matrace.ko`.

Для выгрузки модуля выполняется команда `# rmmod matrace`.

Разработаны необходимые структуры данных (см. листинг 3.1).

Листинг 3.1: Структуры данных

```
1  struct sys_hook {
2      unsigned long long *syscall_table;
3
4      struct sys_hook_ent *head;
5      struct sys_hook_ent *tail;
6  };
7
8  struct sys_hook_ent {
9      unsigned int syscall;
10
11     uintptr_t original;
12     uintptr_t hooked;
13
14     struct sys_hook_ent *next;
15 };
```

При загрузке модуля, то есть в функции `md_init()`, вызывается функция `addr_to_pointer()`, возвращающая адрес таблицы системных вызовов. Далее инициализируется структура `sys_hook`. Вызывается функция `add_hooks()`, которая содержит вызовы функций, инициализирующие функции-перехватчики для `brk`, `mmap` и `munmap`. Функции-перехватчики описаны в файле `hook.c`. На листинге 3.2 представлена функция-перехватчик `brk`. Модификатор `asmlinkage` сообщает компилятору, что функция не должна ожидать найти свои аргументы в регистрах

процессора. `sys_brk_t` — тип данных, определенный в файле `hooks.h` и являющийся указателем на системный вызов.

Листинг 3.2: Функция-перехватчик brk

```
asmlinkage long brk_hook(unsigned long brk)
{
    sys_brk_t sys_brk = (sys_brk_t)sys_hook_get_original(hook,
        __NR_brk);
    long ret = sys_brk(brk);

    log_vma(current, "brk");

    return ret;
}
```

Функция `log_vma()` печатает в журнал ядра суммарный размер областей памяти кучи, стека и анонимных отображений. Каждый системный вызов `brk`, `mmap` и `munmap` сопровождается записью в следующем формате: `[<время>] <название модуля>: <pid вызывающего процесса>: <системный вызов> <размер кучи> <размер стека> <размер анонимных отображений>`.

При выгрузке модуля, то есть в функции `md_exit()`, вызывается функция `sys_hook_free()`, освобождающая память, выделенная под структуру (список) `sys_hook_ent` и структуру `sys_hook`, затем таблица системных вызовов возвращается в исходное состояние.

После завершения процесса, требующего исследования, необходимо прочитать системный журнал и отфильтровать его по искомому pid. Для получения pid процесса можно запустить программу командой `$ ./a.out & $!`. Для получения всех записей исследуемого процесса необходимо выполнить команду `# dmesg | grep "matrace: <pid>:"`. На листинге 3.3 представлен пример результата.

Листинг 3.3: Результат трассировки процесса 71077

```
...
[73149.524945] matrace: 71077: mmap 0 135168 8192
[73149.524954] matrace: 71077: mmap 0 135168 24576
[73149.524983] matrace: 71077: mmap 0 135168 24576
[73149.525103] matrace: 71077: munmap 0 135168 32768
[73149.525191] matrace: 71077: brk 0 135168 32768
[73149.525194] matrace: 71077: brk 135168 135168 32768
...
```

На листингах 4.4–4.10 Приложения А представлен исходный код загружаемого модуля ядра.

## 3.4 Тестирование и отладка загружаемого модуля ядра

Отладка загружаемого модуля ядра с использованием стандартных средств невозможна. В процессе разработки использовалась отладочная печать в журнал ядра: `printk(KERN_DEBUG ‘‘matrace: %s \n‘‘, debug_info)`. В журнал выводились требующие отладки переменные, что позволяло локализовывать ошибки и исправлять их.

Разработанное программное обеспечение было протестировано в полном объеме, найденные ошибки были устранены.

## 3.5 Вывод

В результате разработки загружаемого модуля ядра получено программное обеспечение позволяющее осуществлять мониторинг распределения памяти.

# 4 Исследовательский раздел

## 4.1 Технические характеристики системы

Исследование распределения памяти проводилось на компьютере со следующими характеристиками:

- Процессор Intel Core i5-6200U.
- Оперативная память объемом 6 Гб.
- Операционная система Linux (дистрибутив Ubuntu 20.10 x86-64, ядро версии 5.8.0)

## 4.2 Исследование распределения памяти

Для исследования распределения памяти был разработан набор однопоточных и многопоточных приложений. В данном случае рассматриваются ситуации, при которых потоки завершаются без освобождения памяти. Так как владельцем ресурсов является процесс, выделенная потоку память не возвращается системе до завершения процесса или пока процесс не освободит эту память.

На языке Python разработана программа анализа данных, полученных из загружаемого модуля ядра. Программа получает на вход файл, содержащий часть журнала ядра, строит графики зависимости потребления памяти (куча, стек и анонимные отображения) от времени, рассчитывает максимальное и среднее потребление памяти. Для построения графиков и для расчета статистической информации применяются модули matplotlib и numpy соответственно.

### 4.2.1 Программа с одним выделением памяти

Рассмотрим программы на листингах 4.1 и 4.2. Первая программа выделяет память объемом 128 байтов в основном потоке. Вторая программа выделяет 128 байтов, после чего создает новый поток, который сразу же завершается.

Не смотря на то, что наименьшей адресуемой единицей памяти являются байт и машинное слово, единицей управления памятью в ядре является страница. Размер страницы для 32-разрядное системы обычно составляет 4 килобайта, для 64-разрядной — 8 килобайтов. Таким образом, при запрашивании 128 байтов аллокатор выделяет одну страницу.

Листинг 4.1: `1.c`

```
#include <stdlib.h>

int main()
{
        void *p = malloc(128);

        return 0;
}
```

Листинг 4.2: `2.c`

```
#include <stdlib.h>
#include <pthread.h>

void *routine(void *param)
{
        pthread_exit(0);
}

int main()
{
        void *p = malloc(128);

        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, routine, NULL);
        pthread_join(tid,NULL);

        return 0;
}
```

Графики потребления памяти представлены на рисунке 4.1.

Во второй программе увеличилось максимальное потребление памяти анонимных отображений на 20 килобайтов. Кроме того, на 33% (на 3.67 килобайта) увеличилось среднее потребление памяти кучи. При этом потребление памяти стека остается неизменным.

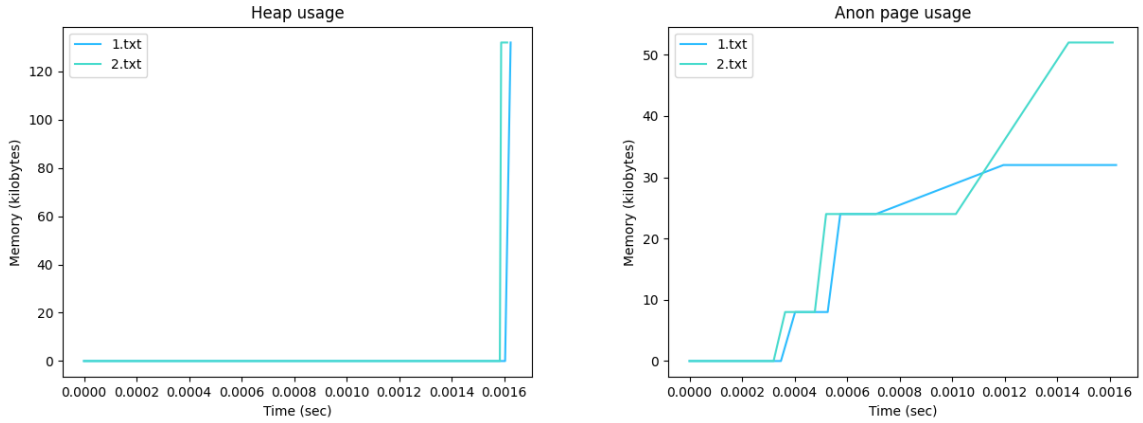


Рис. 4.1: Графики потребления памяти

### 4.2.2 Программа с тремя выделениями памяти

Рассмотрим программы аналогичные предыдущим, за исключением того, что выделение 128 байтов происходит три раза в случае однопоточной программы, два раза в главном потоке и один раз в созданном потоке в случае многопоточной программы.

Графики потребления памяти представлены на рисунке 4.2.

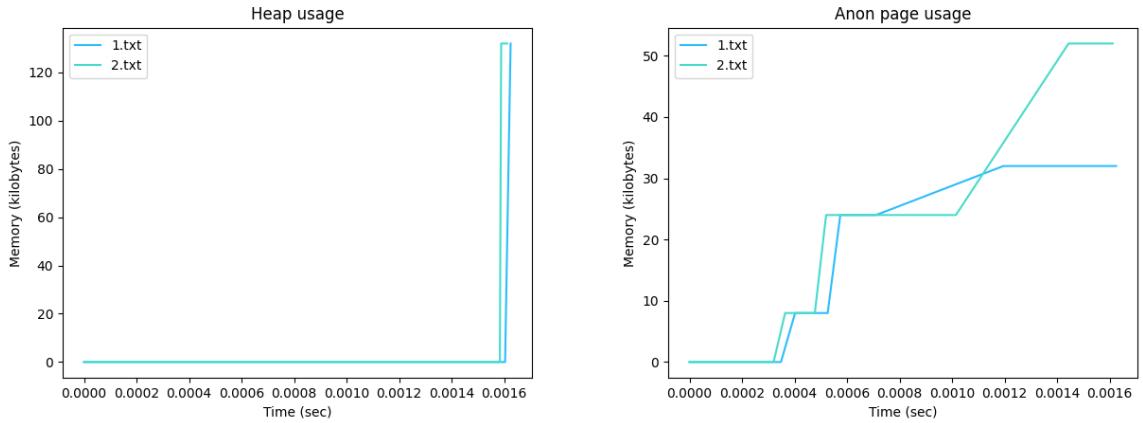


Рис. 4.2: Графики потребления памяти

Полученный результат идентичен результатам предыдущих тестовым программам. Таким образом, выделение памяти в дочернем потоке не отражается на потреблении памяти. Потребление памяти стека также не изменилось. Несмотря на то, что одно из выделений памяти происходит в дочернем процессе, дополнительная страница не выделяется.

### 4.2.3 Программа с двенадцатью выделениями памяти

Увеличим объемы выделяемой памяти. В первой программе последовательно выделим 128, 256, 512, 1024, 2048, 4096 байтов, затем еще 8, 32, 128, 512, 2048, 4096 килобайтов. Во второй программе каждое выделение памяти произведем в отдельном потоке. Таким образом, будет создано 12 потоков.

Графики потребления памяти представлены на рисунке 4.3.

Время выполнения второй программы увеличилось более чем в 2 раза, по сравнению с первой программой. Это связано с временны́ми затратами на создание потоков.

Максимальное потребление памяти кучи в первой и второй программах не отличаются. При этом максимальное потребление памяти анонимных отображений во второй программе увеличилось более чем в 13 раз по сравнению с первой программой. Потребление памяти стека осталось неизменным. Таким образом, можно сделать вывод, что распределение памяти в данном многопоточном приложении выполняется намного менее эффективно в сравнении с однопоточным аналогом. Можно предположить, что с увеличением объемов приложения объем нерационально израсходованной памяти будет увеличиваться. Можно сделать вывод о том, что выделение памяти в 12 потоках приводит к выделению дополнительных страниц, по сравнению с выделение такого же объема памяти в однопоточной программе.

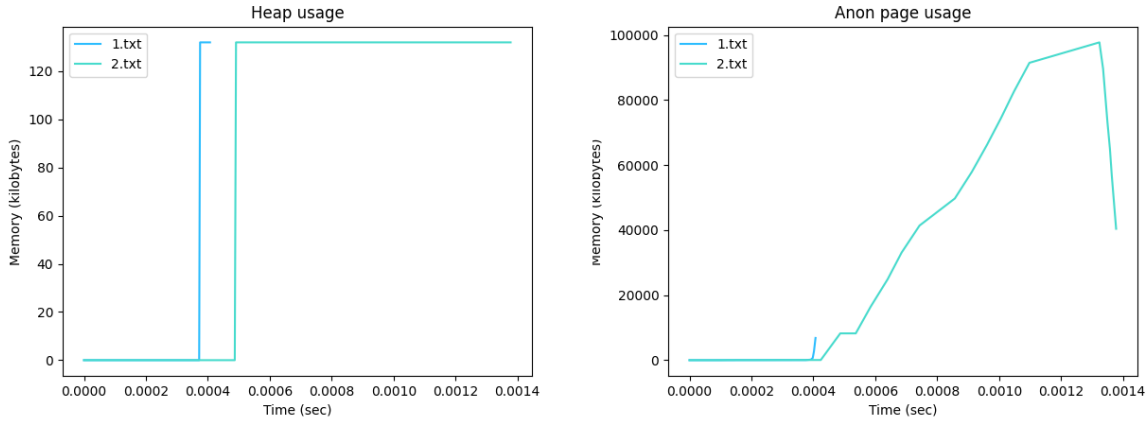


Рис. 4.3: Графики потребления памяти

## 4.3 Вывод

В результате исследования набора тестовых однопоточных и многопоточных программ было установлено, что создание одного потока увеличивает среднее потребление памяти кучи на 3.67 килобайта и максимальное потребление памяти анонимных отображений на 20 килобайтов. Потребление памяти не зависит от места выделения памяти малых объемов при работе с одним дочерним потоком.

С увеличением количества дочерних потоков растет потребление памяти анонимных отображений, из чего можно сделать вывод о том, что выделение областей памяти объемом более 128 килобайт в разных потоках приводит к многократным вызовам `mmap`, выделению дополнительных страниц и неэффективному использованию памяти. При этом потребление памяти кучи изменяется незначительно.

# Заключение

В результате выполнения курсовой работы был разработан загружаемый модуль ядра Linux, позволяющий перехватывать системные вызовы `brk`, `mmap` и `munmap` и выводить информацию о потреблении процессом памяти.

В процессе выполнения курсовой работы были выполнены все поставленные задачи в полном объеме, а именно: проанализированы стандартный аллокатор памяти, коды ядра и методы перехвата системных вызовов, спроектированы структуры данных и алгоритмы, реализован загружаемый модуль ядра, исследовано распределение памяти в однопоточных и многопоточных приложениях.

Было показано, что потребление памяти не зависит от места выделения памяти малых объемов при работе с одним дочерним потоком. Выделение областей памяти объемом более 128 килобайтов в разных потоках приводит к многократным вызовам `mmap` и неэффективному использованию памяти, при этом потребление памяти кучи изменяется незначительно.

# Список использованных источников


1. Лав, Роберт. Ядро Linux: описание процесса разработки, 3-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 496 с. : ил. — Парал. тит. англ.

2. У. Ричард Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. 3-е изд. — СПб.: Питер, 2018. — 944 с.: ил. — (Серия «Для профессионалов»).

3. UNIX изнутри / Ю. Вахалия — СПб.: Питер, 2003. — 844 с.: ил. — (Серия «Классика computer science»)

4. Исходный код ядра Linux версии 5.10 [Электронный ресурс] / Режим доступа — https://elixir.bootlin.com/linux/v5.10/source (дата обращения: 05.11.2020)

# Приложение А. Загружаемый модуль ядра

Листинг 4.3: Makefile

```
obj-m := matrace.o

matrace-objs += module.o
matrace-objs += sys_hook.o
matrace-objs += hooks.o
matrace-objs += log.o

ADDR = $(word 1, $(shell cat /proc/kallsyms | grep "␣sys_call_table"))

# Сборка модуля (по умолчанию)
all default:
        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
           modules

# Загрузка модуля
install: default
        insmod ./matrace.ko syscall_table_addr="$(ADDR)"

# Выгрузка модуля
uninstall:
        rmmod matrace

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

Листинг 4.4: `log.h`

```
#ifndef LOG_H
#define LOG_H

#include <linux/sched.h>

inline void log_vma(struct task_struct *, const char *);
```

```
7
8 #endif
```

Листинг 4.5: log.c

```
1  #include "log.h"
2
3  inline void log_vma(struct task_struct *task, const char *func)
4  {
5      unsigned long heap_size = 0; /* Размер кучи */
6      unsigned long stack_size = 0; /* Размер стека */
7      unsigned long anon_size = 0; /* Размер анонимных отображений */
8
9      struct mm_struct *mm = task->mm;
10     struct vm_area_struct *vma = mm->mmap;
11
12     for (; vma != NULL; vma = vma->vm_next) { /* Цикл по списку
           областей памяти */
13         unsigned long size = vma->vm_end - vma->vm_start;
14
15         if (vma->vm_start <= mm->brk && vma->vm_end >= mm->start_brk) {
16             heap_size += size;
17         } else if (vma->vm_start <= mm->start_stack && vma->vm_end >=
               mm->start_stack) {
18             stack_size += size;
19         } else if (vma->anon_vma != NULL && vma->vm_file == NULL) {
20             anon_size += size;
21         }
22     }
23
24     printk(KERN_INFO "matrace: %d: %s %lu %lu %lu\n", task->pid, func,
           heap_size, stack_size, anon_size);
25 }
```

Листинг 4.6: hooks.h

```
1  #ifndef HOOKS_H
2  #define HOOKS_H
3
4  #include <linux/syscalls.h>
5
6  #include "sys_hook.h"
7  #include "log.h"
8
9  typedef asmlinkage long (*sys_brk_t)(unsigned long);
10 typedef asmlinkage long (*sys_mmap_t)(unsigned long, unsigned long,
       unsigned long, unsigned long, unsigned long, off_t);
11 typedef asmlinkage long (*sys_munmap_t)(unsigned long, size_t);
12
```

```
13 asmlinkage long brk_hook(unsigned long);
14 asmlinkage long mmap_hook(unsigned long, unsigned long, unsigned long,
       unsigned long, unsigned long, off_t);
15 asmlinkage long munmap_hook(unsigned long, size_t);
16
17 #endif
```

Листинг 4.7: hooks.c

```
1  #include "hooks.h"
2
3  extern struct sys_hook *hook;
4
5  /* Функция-перехватчик вызова brk */
6  asmlinkage long brk_hook(unsigned long brk)
7  {
8      sys_brk_t sys_brk = (sys_brk_t)sys_hook_get_original(hook,
           __NR_brk);
9      long ret = sys_brk(brk);
10
11     log_vma(current, "brk");
12
13     return ret;
14 }
15
16 /* Функция-перехватчик вызова mmap */
17 asmlinkage long mmap_hook(unsigned long addr, unsigned long len,
       unsigned long prot,
18                           unsigned long flags, unsigned long fd, off_t
                                pgoff)
19 {
20     sys_mmap_t sys_mmap = (sys_mmap_t)sys_hook_get_original(hook,
           __NR_mmap);
21     long ret = sys_mmap(addr, len, prot, flags, fd, pgoff);
22
23     log_vma(current, "mmap");
24
25     return ret;
26 }
27
28 /* Функция-перехватчик вызова munmap */
29 asmlinkage long munmap_hook(unsigned long addr, size_t len)
30 {
31     sys_munmap_t sys_munmap = (sys_munmap_t)sys_hook_get_original(hook,
           __NR_munmap);
32     long ret = sys_munmap(addr, len);
33
34     log_vma(current, "munmap");
35
```

```
36      return ret;
37  }
```

Листинг 4.8: sys_hook.h

```
1   #ifndef SYS_HOOK_H
2   #define SYS_HOOK_H
3
4   #include <linux/syscalls.h>
5
6   #define CR0_WRITE_PROTECT (1 << 16)
7
8   struct sys_hook {
9       unsigned long long *syscall_table;
10
11      struct sys_hook_ent *head;
12      struct sys_hook_ent *tail;
13  };
14
15  struct sys_hook_ent {
16      unsigned int syscall;
17
18      uintptr_t original;
19      uintptr_t hooked;
20
21      struct sys_hook_ent *next;
22  };
23
24  struct sys_hook *sys_hook_init(uintptr_t);
25  void sys_hook_free(struct sys_hook *);
26
27  bool sys_hook_add(struct sys_hook *, unsigned int, void *);
28
29  uintptr_t sys_hook_get_original(struct sys_hook *, unsigned int);
30
31  #endif
```

Листинг 4.9: sys_hook.c

```
1   #include "sys_hook.h"
2
3   /* Функция, возвращающая регистр CR0 */
4   static uint64_t get_cr0(void)
5   {
6       uint64_t ret;
7
8       __asm__ volatile (
9       "movq␣%%cr0,␣%[ret]"
10      : [ret] "=r"(ret)
```

```
11      );
12
13      return ret;
14  }
15
16  /* Функция, устанавливающая регистр CR0 */
17  static void set_cr0(uint64_t cr0)
18  {
19      __asm__ volatile (
20      "movq %[cr0], %%cr0"
21      :
22      : [cr0] "r"(cr0)
23      );
24  }
25
26  /* Функция, инициализирующая структуру sys_hook */
27  struct sys_hook *sys_hook_init(uintptr_t addr)
28  {
29      struct sys_hook *hook = kmalloc(sizeof(struct sys_hook),
            GFP_KERNEL);
30
31      if (hook == NULL) {
32          printk(KERN_EMERG "matrace: not enough memory\n");
33          return NULL;
34      }
35
36      hook->syscall_table = (unsigned long long *)addr; /* Установка
            адреса системной таблицы */
37      hook->head = NULL;
38      hook->tail = NULL;
39
40      return hook;
41  }
42
43  /* Функция, освобождающая ресурсы */
44  void sys_hook_free(struct sys_hook *hook)
45  {
46      struct sys_hook_ent *cur, *next;
47
48      if (hook == NULL)
49          return;
50
51      set_cr0(get_cr0() & ~CR0_WRITE_PROTECT); /* Модификация CR0 */
52
53      for (cur = hook->head; cur != NULL;) { /* Цикл по всем
            функциям-перехватчикам */
54          hook->syscall_table[cur->syscall] = (unsigned long
                long)cur->original;
```

```
55
56          next = cur->next;
57          kfree(cur);
58          cur = next;
59      }
60
61      set_cr0(get_cr0() | CR0_WRITE_PROTECT); /* Модификация CR0 */
62
63      kfree(hook);
64  }
65
66  /* Функция, инициализирующая функцию-перехватчик */
67  bool sys_hook_add(struct sys_hook *hook, unsigned int syscall, void
        *hook_func)
68  {
69      struct sys_hook_ent *entry = kmalloc(sizeof(struct sys_hook_ent),
            GFP_KERNEL);
70
71      if (entry == NULL) {
72          printk(KERN_EMERG "matrace: not enough memory\n");
73          return false;
74      }
75
76      entry->syscall = syscall; /* Установка идентификатора вызова */
77      entry->original = hook->syscall_table[syscall]; /* Установка адреса
            оригинального вызова */
78      entry->hooked = (uintptr_t)hook_func; /* Установка указателя на
            функцию-перехватчик */
79      entry->next = NULL;
80
81      set_cr0(get_cr0() & ~CR0_WRITE_PROTECT); /* Модификация CR0 */
82      hook->syscall_table[syscall] = (unsigned long long)entry->hooked;
            /* Модификация системной таблицы */
83      set_cr0(get_cr0() | CR0_WRITE_PROTECT); /* Модификация CR0 */
84
85      if (hook->head == NULL)
86          hook->head = hook->tail = entry;
87      else {
88          hook->tail->next = entry;
89          hook->tail = entry;
90      }
91
92      return true;
93  }
94
95  /* Функция, возвращающая адрес оригинального системного вызова */
96  uintptr_t sys_hook_get_original(struct sys_hook *hook, unsigned int
        syscall)
```

```
97  {
98      struct sys_hook_ent *cur = hook->head;
99
100     for (; cur != NULL; cur = cur->next) { /* Цикл по списку
            функций-перехватчиков */
101         if (cur->syscall == syscall)
102             return cur->original;
103     }
104
105     return 0;
106 }
```

Листинг 4.10: module.c

```
1  #include <linux/module.h>
2
3  #include "hooks.h"
4
5  MODULE_LICENSE("GPL");
6  MODULE_AUTHOR("Mikael␣Novikov");
7  MODULE_DESCRIPTION("Memory␣allocation␣trace␣module");
8
9  /* Аргумент, содержащий адрес таблицы системных вызовов */
10 static char *syscall_table_addr = NULL;
11 module_param(syscall_table_addr, charp, 0);
12 MODULE_PARM_DESC(syscall_table_addr, "Base␣address␣of␣the␣syscall␣
       table");
13
14 struct sys_hook *hook;
15
16 /* Преобразование адреса к указателю */
17 static uintptr_t addr_to_pointer(const char *str)
18 {
19     uintptr_t addr = 0;
20
21     for (; *str != '\0'; ++str) {
22         addr *= 16;
23
24         if (*str >= '0' && *str <= '9')
25             addr += (*str - '0');
26         else if (*str >= 'a' && *str <= 'f')
27             addr += (*str - 'a') + 10;
28         else
29             return 0;
30     }
31
32     return addr;
33 }
34
```

```
35 static inline void add_hooks(void)
36 {
37     sys_hook_add(hook, __NR_brk, (void *)brk_hook);
38     sys_hook_add(hook, __NR_mmap, (void *)mmap_hook);
39     sys_hook_add(hook, __NR_munmap, (void *)munmap_hook);
40 }
41
42 static int __init md_init(void)
43 {
44     uintptr_t addr;
45
46     /* Проверка аргумента адреса системной таблицы */
47     if (syscall_table_addr == NULL || syscall_table_addr[0] == '\0') {
48         printk(KERN_EMERG "matrace: failed to get syscall table\n");
49         return 1;
50     }
51
52     addr = addr_to_pointer(syscall_table_addr); /* Преобразование
           адреса в указатель */
53     if (addr == 0) {
54         printk(KERN_EMERG "matrace: invalid x64 syscall address\n");
55         return 1;
56     }
57
58     hook = sys_hook_init(addr); /* Инициализация структуры sys_hook */
59     if (hook == NULL) {
60         printk(KERN_EMERG "matrace: failed to initialize sys_hook\n");
61         return 1;
62     }
63
64     add_hooks(); /* Инициализация функций-перехватчиков */
65
66     printk(KERN_INFO "matrace: module loaded\n");
67
68     return 0;
69 }
70
71 static void __exit md_exit(void)
72 {
73     sys_hook_free(hook); /* Освобождение памяти */
74
75     printk(KERN_INFO "matrace: module unloaded\n");
76 }
77
78 module_init(md_init);
79 module_exit(md_exit);
```