



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:***  
**«Мониторинг процессов»**

Студент группы **ИУ7-76Б**

\_\_\_\_\_  
(Подпись, дата) **Мансуров В. М.**  
(Фамилия И.О.)

Руководитель

\_\_\_\_\_  
(Подпись, дата) **Рязанова Н. Ю.**  
(Фамилия И.О.)

2023 г.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ</b>	<b>5</b>
<b>1 Аналитическая часть</b>	<b>6</b>
1.1 Постановка задачи . . . . .	6
1.2 Анализ структур и функций ядра . . . . .	7
1.2.1 Структура процесса . . . . .	7
1.2.2 Адресное пространство процесса . . . . .	14
1.2.3 Сигналы . . . . .	16
1.2.4 Семафоры . . . . .	19
1.2.5 Сегмента разделяемой памяти . . . . .	25
1.2.6 Программные каналы . . . . .	29
1.3 Подход перехвата функций . . . . .	30
1.3.1 kprobes . . . . .	30
1.3.2 ftrace . . . . .	31
1.4 Простанство ядра и пользователя . . . . .	33
1.5 Виртуальная файловая система /proc . . . . .	34
<b>2 Конструкторская часть</b>	<b>38</b>
2.1 Последовательность действий . . . . .	38
2.2 Разработка алгоритмов . . . . .	39
<b>3 Технологическая часть</b>	<b>46</b>
3.1 Выбор языка и среды программирования . . . . .	46
3.2 Реализация загружаемого модуля . . . . .	46
<b>4 Исследовательский раздел</b>	<b>68</b>
4.1 Технические характеристики . . . . .	68
4.2 Исследование работы программы . . . . .	68

<b>ЗАКЛЮЧЕНИЕ</b>	<b>73</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>74</b>

# ВВЕДЕНИЕ

В настоящее время UNIX-подобными операционными системами являются одними из самых популярных в мире. На персональных компьютерных за последние 10 лет такие системы встречаются у 20% пользователей [1], а на мобильных устройствах около 70% [2]. Для пользователей данных систем важно предоставлять информацию о выполнении процесса, который постоянно используют память, посылают и принимают сигналы, семафоры, сегменты разделяемой памяти и программные каналы. Особое внимание уделяется операционным системам Linux, так как ядро Linux возможно изучить, благодаря тому что оно имеет открытый исходный код.

Целью курсовой работы является разработка загружаемого модуля ядра, предоставляющего информацию о приоритете, времени выполнения и простоя, выделенной виртуальной памяти, сегментах разделяемой памяти, программных каналов, семафорах и сигналах процессов.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- провести анализ структур и функций, предоставляющих возможность реализовать поставленную задачу;
- разработать алгоритмы и структуру загружаемого модуля ядра, обеспечивающего отслеживание процессов.

# 1 Аналитическая часть

## 1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу необходимо разработать загрузаемый модуль ядра для получения информации о приоритетах, времени выполнения и простоя, выделенной виртуальной памяти, сегментах разделяемой памяти, программных каналов, семафорах и сигналов процессов. Для поставленной задачи необходимо:

- провести анализ структур и функций ядра, предоставляющих возможность реализовать поставленную задачу;
- провести анализ подходов к перехвату функций;
- провести анализ методы передачи информации из пространства ядра в пространство пользователя и наоборот;
- разработать алгоритмы и структуру загружаемого модуля ядра, в соответствии с поставленной задачей;
- спроектировать и реализовать загрузаемый модуль ядра;
- исследовать работу реализованного модуля ядра.

К разрабатываемой программе предъявляются следующие требования:

- взаимодействие с загружаемым модулем должно происходить из пространства пользователя;
- необходимо передавать данные из пространства ядра в пространство пользователя или наоборот.

## 1.2 Анализ структур и функций ядра

### 1.2.1 Структура процесса

Каждый процесс в ядре описывается структурой `struct task_struct`, которая содержит информацию, необходимую для управления процессом и называется дескриптором процесса, описана в файле `<linux/sched.h>`.

Множество процессов в Linux-системе представляет собой совокупность структур `struct task_struct`, которые взаимосвязаны по средством кольцевого двусвязанного списка. Максимальное количество процессов, которое может быть создано, ограничивается только объемом физической памяти. В листинге 1.1 представлены необходимые для решения задачи фрагменты структуры `task_struct` [3].

Листинг 1.1 – Структура `struct task_struct`

```
1 struct task_struct {
2     unsigned int    __state;
3
4     unsigned int    flags;
5
6     int             prio;
7     int             static_prio;
8     int             normal_prio;
9     unsigned int    rt_priority;
10
11     struct sched_entity    se;
12     struct sched_rt_entity    rt;
13     struct sched_dl_entity    dl;
14     const struct sched_class    *sched_class;
15
16     struct sched_statistics    stats;
17
18     unsigned int    policy;
```

Листинг 1.2 – Структура struct task\_struct

```

1
2  struct sched_info      sched_info;
3
4  struct list_head       tasks;
5
6  struct mm_struct       *mm;
7  struct mm_struct       *active_mm;
8
9  int                    exit_state;
10 int                    exit_code;
11 int                    exit_signal;
12
13 pid_t                  pid;
14 pid_t                  tgid;
15
16 struct task_struct     __rcu    *real_parent;
17 /* Recipient of SIGCHLD, wait4() reports: */
18 struct task_struct     __rcu    *parent;
19 /* Children/sibling form the list of natural children:*/
20 struct list_head       children;
21 struct list_head       sibling;
22 struct task_struct     *group_leader;
23 /* PID/PID hash table linkage. */
24 struct pid             *thread_pid;
25 struct hlist_node      pid_links[PIDTYPE_MAX];
26 struct list_head       thread_group;
27 struct list_head       thread_node;
28
29 u64                    utime;
30 u64                    stime;
31 u64                    gtime;
32 u64                    start_time;
33 u64                    start_boottime;

```

Листинг 1.3 – Структура struct task\_struct

```

1      char                comm[TASK_COMM_LEN];
2
3  #ifdef CONFIG_SYSVIPC
4      struct sysv_sem      sysvsem;
5      struct sysv_shm      sysvshm;
6  #endif
7
8      /* Signal handlers: */
9      struct signal_struct  *signal;
10     struct sighand_struct __rcu *sighand;
11     struct sigpending      pending;
12 };

```

Подробное описание представленного фрагмента структуры `task_struct`.

Состояние процесса определяется следующими членами, значения которых представлены в листинге 1.4:

- 1) `__state` — состояние процесса;
- 2) `exit_state` — состояние завершения процесса.

Листинг 1.4 – Макросы состояния процесса

```

1  /* Значение __state: */
2  #define TASK_RUNNING          0x00000000
3  #define TASK_INTERRUPTIBLE    0x00000001
4  #define TASK_UNINTERRUPTIBLE  0x00000002
5  #define __TASK_STOPPED        0x00000004
6  #define __TASK_TRACED         0x00000008
7  /* Значения exit_state: */
8  #define EXIT_DEAD              0x00000010
9  #define EXIT_ZOMBIE            0x00000020
10 #define EXIT_TRACE              (EXIT_ZOMBIE | EXIT_DEAD)
11 /* Значение __state: */
12 #define TASK_PARKED             0x00000040
13 #define TASK_DEAD               0x00000080

```



# Листинг 1.5 – Макросы состояния процесса

```

1 #define TASK_WAKEKILL          0x00000100
2 #define TASK_WAKING            0x00000200
3 #define TASK_NOLOAD            0x00000400
4 #define TASK_NEW                0x00000800
5 #define TASK_RTLOCK_WAIT       0x00001000
6 #define TASK_FREEZABLE         0x00002000
7 #define __TASK_FREEZABLE_UNSAFE (0x00004000 *
    IS_ENABLED(CONFIG_LOCKDEP))
8 #define TASK_FROZEN            0x00008000
9 #define TASK_STATE_MAX         0x00010000
10 #define TASK_ANY               (TASK_STATE_MAX-1)
11 #define TASK_KILLABLE          (TASK_WAKEKILL |
    TASK_UNINTERRUPTIBLE)
12 #define TASK_STOPPED           (TASK_WAKEKILL | __TASK_STOPPED)
13 #define TASK_TRACED            __TASK_TRACED
14 #define TASK_IDLE              (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
15 #define TASK_NORMAL            (TASK_INTERRUPTIBLE |
    TASK_UNINTERRUPTIBLE)
16 #define TASK_REPORT            (TASK_RUNNING | TASK_INTERRUPTIBLE | \
17     TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
18     __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
19     TASK_PARKED)
20
21 #define task_is_running(task)   (READ_ONCE((task)->__state)
    == TASK_RUNNING)
22
23 #define task_is_traced(task)    ((READ_ONCE(task->jobctl) &
    JOBCTL_TRACED) != 0)
24 #define task_is_stopped(task)  ((READ_ONCE(task->jobctl) &
    JOBCTL_STOPPED) != 0)
25 #define task_is_stopped_or_traced(task) ((READ_ONCE(task->jobctl)
    & (JOBCTL_STOPPED | JOBCTL_TRACED)) != 0)

```

Идентификатор процесса и члены, представляющий родство процесса:

- 1) `pid` — уникальный идентификатор процесса;
- 2) `ppid` — идентификатор процесса-родителя;
- 3) `tgid` — идентификатор лидера группы потоков;
- 4) `real_parent` — указывает на родительский процесс, если родительский процесс завершился, то указывает на процесс, `pid` которого равен 1;
- 5) `parent` — родительский процесс;
- 6) `children` — двусвязанный список дочерних процессов;
- 7) `sibling` — двусвязанный список, используемые для связанности дочерних процессов с родительским процессом, которая представлена на рисунке 1.1;
- 8) `group_leader` — процесс-лидер группы процессов.

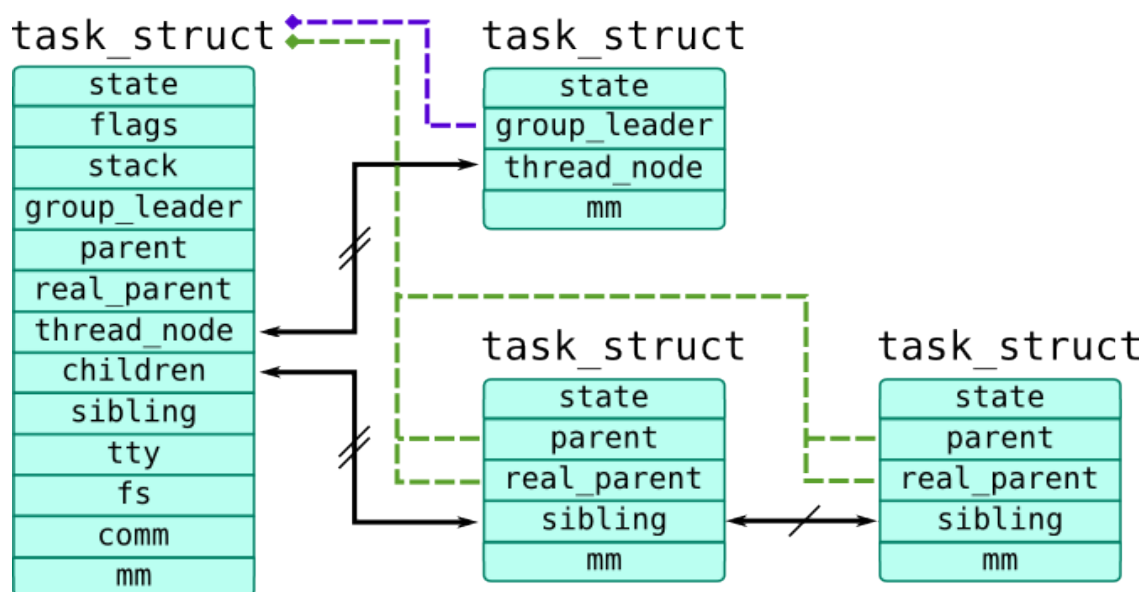


Рис. 1.1 – Связанность дочерних процессов с родительским процессом

Приоритет и алгоритм планировщика задач, описываются в следующих членах:

- 1) `prio` — приоритет процесса, которое используется планировщиком задач при выборе процесса. Чем ниже значение данной переменной, тем выше приоритет процесса. Диапазон значений от 0 до 139, то есть `MAX_PRIO`. Данный диапазон делится на два интервала:

- 0 – 99 — приоритет процессов в реальном времени;
- 100 – 139 — приоритет обычных процессов, по умолчанию 120;

Особое внимание необходимо уделить процессу с приоритетом 0 — `migration`, который перераспределяет процессы между ядрами процессора. У таких процессов алгоритм планирования устанавливается FIFO, по причине того, что данный процесс имеет наивысший приоритет и ему необходимо выполниться от начала и до конца;

- 2) `static_prio` — приоритет процесса, который не изменяется при работе планировщика задач, однако оно может быть изменено с использованием системного вызова `nice`;
- 3) `normal_prio` — приоритет процесса, который зависит от статического приоритета и алгоритма планировщика задач;
- 4) `rt_proirity` — приоритет процессов в реальном времени;
- 5) `policy` — алгоритма планирования, значение которого представлены в листинге 1.6.

Листинг 1.6 – Макросы значений `policy`

```
1 #define SCHED_NORMAL    0
2 #define SCHED_FIFO      1
3 #define SCHED_RR         2
4 #define SCHED_BATCH      3
5 #define SCHED_IDLE       5
```

Время выполнения описывается следующими членами:

- 1) `utime` — время процесса, проведенное в режиме пользователя;
- 2) `stime` — время процесса, затраченное на выполнение системных вызовов;
- 3) `start_time` — время создания процесса;
- 4) `start_boottime` — время ожидания процесса.

### **struct sched\_info**

`struct sched_info` — структура, которая предоставляет информацию о планировании процесса, представлена в листинге 1.7.

Листинг 1.7 – `struct sched_info`

```
1 struct sched_info {  
2 #ifdef CONFIG_SCHED_INFO  
3     unsigned long      pcount;  
4     unsigned long long  run_delay;  
5     unsigned long long  last_arrival;  
6     unsigned long long  last_queued;  
7 #endif  
8 };
```

Подробное описание представленной структуры `sched_info`:

- 1) `pcount` — количество запусков процесса на исполнение центральным процессом;
- 2) `run_delay` — количество времени, проведенного в ожидании на исполнение;
- 3) `last_arrive` — время последнего запуска процесса на исполнение центральным процессором;
- 4) `last_queued` — время последнего процесса в очередь на исполнение центральным процессором.

### 1.2.2 Адресное пространство процесса

`struct mm_struct` — структура, которая описывает адресное пространство процесса и описана в файле `<linux/mm_types.h>`. У `task_struct` есть два указателя `mm` и `active_mm` на структуру `mm_struct`, используемые во время выполнения процесса. Для обычных процессов значения этих двух переменных-указателей одинаковы. Однако потоки не владеют адресным пространством, поэтому `mm` имеет значение `NULL`, а `active_mm` указывает на дескриптор адресного пространства процесса, создавшего поток. Фрагмент этой структуры представлен в листинге 1.8.

Листинг 1.8 – `struct mm_struct`

```
1 struct mm_struct {
2     struct vm_area_struct *mmap;    /* list of VMAs */
3     struct rb_root mm_rb;
4     ...
5     unsigned long mmap_base;
6     ...
7     unsigned long task_size;        /* size of task vm space */
8     unsigned long highest_vm_end;    /* highest vma end address */
9     pgd_t * pgd;
10    ...
11    atomic_t mm_users;
12    atomic_t mm_count;
13    ...
14    int map_count;                   /* number of VMAs */
15    spinlock_t page_table_lock;
16    struct rw_semaphore mmap_lock;
17    ...
18    struct list_head mmlist;
19    ...
20    unsigned long total_vm; /* Total pages mapped */
21    unsigned long locked_vm; /* Pages that have PG_mlocked set */
```

Листинг 1.9 – struct mm\_struct

```

1  atomic64_t    pinned_vm; /* Refcount permanently increased */
2  unsigned long data_vm; /* VM_WRITE & ~VM_SHARED & ~VM_STACK */
3  unsigned long exec_vm; /* VM_EXEC & ~VM_WRITE & ~VM_STACK */
4  unsigned long stack_vm; /* VM_STACK */
5  ...
6  unsigned long start_code, end_code, start_data, end_data;
7  unsigned long start_brk, brk, start_stack;
8  unsigned long arg_start, arg_end, env_start, env_end;
9  ...
10 /*
11  * Special counters, in some configurations protected by the
12  * page_table_lock, in other configurations by being atomic.
13  */
14 struct mm_rss_stat rss_stat;
15 ...
16 unsigned long flags; /* Must use atomic bitops to access */
17 };

```

struct vm\_area\_struct — структура, которая описывает непрерывную область адресного пространства процесса. У объекта mm\_struct содержит указатель на структуру vm\_area\_struct. Фрагмент этой структуры представлен в листинге 1.10.

Листинг 1.10 – struct vm\_area\_struct

```

1 struct vm_area_struct {
2
3     unsigned long vm_start; /* Our start address within vm_mm. */
4     unsigned long vm_end; /* The first byte after our end
5                             address within vm_mm. */
6
7     /* linked list of VM areas per task, sorted by address */
8     struct vm_area_struct *vm_next, *vm_prev;
9 };

```

Листинг 1.11 – struct vm\_area\_struct

```

1  ...
2  struct mm_struct *vm_mm; /* The address space we belong to. */
3  ...
4  unsigned long vm_flags; /* Flags, see mm.h. */
5  ...
6  /* Serialized by mmap_lock &
7   * page_table_lock */
8  struct list_head anon_vma_chain;
9  struct anon_vma *anon_vma; /* Serialized by page_table_lock
   */
10
11 /* Function pointers to deal with this struct. */
12 const struct vm_operations_struct *vm_ops;
13
14 /* Information about our backing store: */
15 /* Offset (within vm_file) in PAGE_SIZE units */
16 unsigned long vm_pgoff;
17 struct file * vm_file; /* File we map to (can be NULL). */
18 void * vm_private_data; /* was vm_pte (shared mem) */
19 } __randomize_layout;

```

### 1.2.3 Сигналы

Сигнал — способ информирования процесса ядром о происшествии какого-то события. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло. Если возникло несколько однотипных событий, процессу будет подан только один сигнал.

`struct signal_struct` — структура, которая описывает сигнал процесса. Объект `task_struct` содержит указатель на данную структуру. Данная структура представлена в листинге 1.12.

Листинг 1.12 – struct sighand\_struct

```

1 struct signal_struct {
2     refcount_t      sigcnt;
3     atomic_t        live;
4     int             nr_threads;
5     struct list_head thread_head;
6
7     /* current thread group signal load-balancing target: */
8     struct task_struct *curr_target;
9
10    struct sigpending shared_pending;
11
12    struct hlist_head multiprocess;
13
14    int              group_exit_code;
15    /* notify group_exec_task when notify_count is less or equal
16       to 0 */
17    int              notify_count;
18    struct task_struct *group_exec_task;
19
20    /* thread group stop support, overloads group_exit_code too */
21    int              group_stop_count;
22    unsigned int      flags; /* see SIGNAL_* flags below */
23 } __randomize_layout;

```

**struct sighand\_struct** — структура, которая описывает обработчики сигналов процесса. Эта структура описана в файле `<linux/sched/signal.h>`. У объекта **task\_struct** содержит указатель на данную структуру. Эта структура представлена в листинге 1.13.

**struct k\_sigaction** — структура, которая описывает обработчик сигнала процесса. Эта структура представлена в листинге 1.14.



Листинг 1.13 – struct sighand\_struct

```

1 struct sighand_struct {
2     spinlock_t      siglock;
3     refcount_t      count;
4     wait_queue_head_t  signalfd_wqh;
5     struct k_sigaction  action[_NSIG];
6 };

```

Листинг 1.14 – struct k\_sigaction и struct sigaction

```

1 struct k_sigaction {
2     struct sigaction sa;
3 #ifdef __ARCH_HAS_KA_RESTORER
4     __sigrestore_t ka_restorer;
5 #endif
6 };
7
8 struct sigaction {
9     __sighandler_t sa_handler;
10    unsigned long sa_flags;
11    __sigrestore_t sa_restorer;
12    sigset_t sa_mask;
13 };

```

Для работы с сигналами процессу предоставляется следующие системные вызовы.

Системный вызов `signal()` устанавливает обработчик на указанный сигнал. Важно отметить, что процесс имеет 64 сигналов и у каждого из них есть обработчики по умолчанию. В листинге 1.15 представлен данный системный вызов.

Листинг 1.15 – Системный вызов signal()

```

1 int signal(int sig, __sighandler_t handler);

```

В качестве параметров передаются:

- **sig** — номер сигнала;
- **handler** — адрес функции, которая должна быть выполнена при поступлении указанного сигнала.

Системный вызов возвращает указатель на предыдущий обработчик данного сигнала, который можно использовать для восстановления обработчика. Вместо адреса обработчика можно указать 0 или 1. Если был указан 0, то при поступлении сигнала выполнение будет прервано. Если была указана 1, то сигнал будет проигнорирован.

Системный вызов `kill()` отправляет сигнал указанному процессу. В листинге 1.16 представлен данный системный вызов.

Листинг 1.16 – Системный вызов `kill()`

```
1 int kill(pid_t pid, int sig);
```

В качестве параметров передаются:

- **pid** — уникальный идентификатор процесса, которому будет отправлен сигнал;
- **sig** — номер сигнала.

Если вместо **pid** указать 0, то сигнал будет послан всем процессам. Если вместо **pid** указать -1, то ядро передает сигнал всем процессам, идентификатор пользователя которых равен идентификатору текущего процесса, который посылает сигнал.

#### 1.2.4 Семафоры

Linux поддерживает наборы считавших семафоров, которые семантически представлены в системе массивами и доступ к отдельному семафору осуществляется по номеру, начиная с 0. Основным свойством набора семафоров является возможность одной неделимой операцией изменить значения всех или части семафоров набора

Для описания всех семафоров в ядре имеется таблица семафоров, в

которой отслеживаются все созданные наборы семафоров, структура `struct semid_ds`. Данная структура представлена в листинге 1.17.

Листинг 1.17 – `struct semid_ds`

```
1 struct semid_ds {
2     struct ipc_perm sem_perm;
3     __kernel_old_time_t sem_otime;
4     __kernel_old_time_t sem_ctime;
5     struct sem *sem_base;
6     struct sem_queue *sem_pending;
7     struct sem_queue **sem_pending_last;
8     struct sem_undo *undo;
9     unsigned short sem_nsems;
10 };
```

Если несколько семафоров объединены в массив, то этот массив является набором семафоров. `structs sem_array` — структура, которая описывает набор семафоров. Данная структура представлена в листинге 1.18.

Листинг 1.18 – `struct sem_array`

```
1 struct sem_array {
2     struct kern_ipc_perm sem_perm;
3     time64_t sem_ctime;
4     struct list_head pending_alter;
5     struct list_head pending_const;
6     struct list_head list_id;
7     int sem_nsems;
8     int complex_count;
9     unsigned int use_global_lock;
10
11     struct sem sems[];
12 } __randomize_layout;
```

В двух структурах есть схожие поля, которые имеют одно и тоже значение. Данные поля описаны ниже:

- 1) `sem_perm` — информация о доступе к множеству семафоров, включая права доступа, и создателе семафора;
- 2) `sem_otime` — время последней операции `semop()`;
- 3) `sem_ctime` — время последнего изменения структуры;
- 4) `sem_base` — указатель на первый семафор в массиве;
- 5) `sem_nsems` — количество семафоров в массиве.

В `struct semid_ds` и `struct sem_array` есть указатель на структуру `struct sem`, которая описывает семафор. Данная структура представлена в листинге 1.19.

Листинг 1.19 – `struct sem`

```
1 struct sem {  
2     int semval;  
3     struct pid      *sempid;  
4     spinlock_t      lock;  
5     struct list_head pending_alter;  
6     struct list_head pending_const;  
7     time64_t        sem_otime;  
8 } _____cacheline_aligned_in_smp;
```

Подробнее описание полей структуры:

- 1) `sempid` — идентификатор процесса, проделавшего последнюю операцию;
- 2) `semval` — текущее значение семафора;
- 3) `sem_otime` — время последней операции `semop()`.

В Linux имеются системные вызовы для создания, изменение управляющих параметров и уничтожения набора семафора, выполнения операций на семафорах. Данные системные вызовы пойманы ниже.

Системный вызов `semget()` создает новый набор семафоров или открывает уже существующий набор семафоров. Данный системный вызов представлен в листинге 1.20.

### Листинг 1.20 – Системный вызов `semget()`

```
1 int semget(key_t key, int numb_sem, int flag);
```

В качестве параметров принимает:

- `key` — целочисленное значение, позволяющее несвязанным процессам обращаться к одному и тому же семафору;
- `nsems` — количество семафоров, максимально значение которого равно значению макроса `SEMMSL`;
- `semflg` — результат побитого сложения прав доступа к семафору и `IPC_CREATE`.

В случае успешного завершения системный вызов возвращает идентификатор семафора, а в случае неудачи возвращается -1. Существует особое значение ключа семафора — `IPC_PRIVATE`, которое предназначено для создания семафора, доступ к которому получает только сам процесс и процессы, порожденные процессом, создавшим семафор.

Допустимые флаги:

- `IPC_CREAT` создает набор семафоров, если его еще не было в системе;
- `IPC_EXCL` при использовании вместе с `IPC_CREAT` вызывает ошибку, если семафор уже существует. Сам по себе `IPC_EXCL` бесполезен, но вместе с `IPC_CREAT` он дает средство гарантировать, что ни одно из существующих множеств семафоров не открыто для доступа.

Системный вызов `semctl()` используется для осуществления управления множеством семафоров. Данный системный вызов представлен в листинге 1.21.

### Листинг 1.21 – Системный вызов `semctl()`

```
1 int semctl(int semid, int semnum, int cmd, union semun arg);
```

В качестве параметров передаются:

- `semid` — идентификатор набора семафоров;
- `semnum` — номер семафора в наборе семафоров;

- `cmd` — команда, которая будет выполнена над набором семафоров;
- `arg` — объединение `semun`.

Доступные команды над набором семафоров:

- 1) `IPC_STAT` — возвращается структура `semid_ds` для множества и запоминает ее по адресу аргумента `buf` в объединении `semun`;
- 2) `IPC_SET` — устанавливает значение элемента `ipc_perm` структуры `semid_ds` для множества;
- 3) `IPC_RMID` — удаляет множество из ядра;
- 4) `GET_ALL` — возвращает значение всех семафоров множества, целые значения запоминаются в массиве элементов `unsigned short`, на который указывает член объединения `array`;
- 5) `GETNCNT` — выдает число процессов, ожидающих ресурсов в данный момент;
- 6) `GETPID` — возвращает PID процесса, выполнившего последний вызов `semop()`;
- 7) `GETVAL` — возвращает значение одного семафора из множества;
- 8) `GETZCNT` — возвращает число процессов, ожидающих 100% освобождения ресурса;
- 9) `SETALLM` — устанавливает значения семафоров множества, взятые из элемента `array` объединения `args`;
- 10) `SETVAL` — устанавливает значение конкретного семафора множества как элемент `val` объединения `args`.

Объединение `union semun` представлено в листинге 1.22.

Листинг 1.22 – `union semun`

```
1 union semun {  
2     int val;  
3     struct semid_ds *buf;  
4     ushort *array;  
5     struct seminfo *__buf;
```

Системный вызов `semop()` выполняет операцию на наборе семафоров. Данный системный вызов представлен в листинге 1.23.

Листинг 1.23 – Системный вызов `semop()`

```
1 int semop(int semid, struct sembuf *sop, unsigned nsop);
```

В качестве параметров передаются:

- `semid` — идентификатор набора семафоров;
- `sop` — указатель на структуру операции на семафоре;
- `nsop` — количество операций в этом массиве.

Структура `struct sembuf`, которая предназначена для описания выполнения операции на семафоре. Данная структура представлена в листинге 1.24.

Листинг 1.24 – `struct sembuf`

```
1 struct sembuf {  
2     unsigned short    sem_num ;  
3     short            sem_op ;  
4     short            sem_flg ;  
5 };
```

Подробное описание данной структуры:

- 1) `sem_num` — номер семафора в наборе семафоров;
- 2) `sem_op` — выполняемая операция (положительное, отрицательное число или нуль);
- 3) `sem_flg` — флаги.

Если `sem_op` отрицателен, то его значение вычитается из семафора. Это соответствует получению ресурсов, которые контролирует семафор. Если `IPC_NOWAIT` не установлен, то вызывающий процесс блокируется, пока семафор не выдаст требуемое количество ресурсов. Если `sem_op` положителен, то его значение добавляется к семафору. Это соответствует возвращению ресурсов множеству семафоров приложения. Если `sem_op` равен 0, то вызывающий

процесс будет блокирован, пока значение семафора не станет 0. Это соответствует ожиданию того, что ресурсы будут использованы на 100%.

### 1.2.5 Сегмента разделяемой памяти

Разделяемая память является средством передачи информации от процесса к процессу. Разделяемая память (сегменты разделяемой памяти) была разработана для сокращения времени передачи сообщений за счет исключения необходимости копировать текст сообщения из пространства пользователя в пространство ядра. Это обеспечивается за счет возможности подключения разделяемого сегмента к адресному пространству процесса, а именно за счет возможности получения процессом указателя на разделяемый сегмент. Аналогично программным каналам разделяемые сегменты создаются в разделяемой памяти, которой является область данных ядра системы. В отличие от программных каналов разделяемая память не имеет встроенных средств взаимного исключения и, как правило, используется совместно с семафорами. Аналогично семафорам дескрипторы всех разделяемых сегментов находятся в системной таблице разделяемой памяти ядра системы.

Структура `struct shmid_ds`, которая описывает системную таблицу разделяемой памяти. Данная структура представлена в листинге 1.25.

Листинг 1.25 – `struct shmid_ds`

```
1 struct shmid_ds {  
2     struct ipc_perm    shm_perm;  
3     int                shm_segsz;  
4     __kernel_old_time_t shm_atime;  
5     __kernel_old_time_t shm_dtime;  
6     __kernel_old_time_t shm_ctime;  
7     __kernel_ipc_pid_t  shm_cpid;  
8     __kernel_ipc_pid_t  shm_lpid;  
9     unsigned short      shm_nattch;  
10 };
```

Подробное описание полей структуры:



- 1) `shm_perm` — информация о доступе к разделяемой памяти, включая права доступа и создателя сегмента разделяемой памяти;
- 2) `shm_segsz` — раздел сегмента в байтах;
- 3) `shm_atime` — время последнего подключения;
- 4) `shm_dtime` — время последнего отключения;
- 5) `shm_ctime` — время последнего изменения;
- 6) `shm_cpid` — идентификатор процесса-создателя;
- 7) `shm_lpid` — идентификатор последнего пользователя;
- 8) `shm_nattch` — число процессов, привязанных к сегменту разделяемой памяти.

В Linux имеются системные вызовы для создания разделяемой памяти, изменения управляющих параметров созданного сегмента, подключения сегмента к адресному пространству процесса, т.е. получения указателя на него и отключения сегмента разделяемой памяти от адресного пространства процесса.

Системный вызов `shmget()` создает новый разделяемый сегмент или, если сегмент уже существует, то права доступа подтверждаются. Системный вызов возвращает в случае успеха идентификатор сегмента разделяемой памяти, иначе -1. Данный системный вызов представлен в листинге 1.26.

Листинг 1.26 – Системный вызов `shmget()`

```
1 int shmget(key_t key, size_t size, int flag);
```

В качестве параметров принимаются:

- `key` — целочисленное значение, позволяющее несвязанным процессам обращаться к одному и тому же сегменту разделяемой памяти;
- `size` — размер памяти в байтах, которое будет выделено;
- `flag` — результат побитового сложения прав доступа к сегменту разделяемой памяти и `IPC_CREATE`.

Доступные флаги при создании сегмента разделяемой памяти:

- 1) `IPC_CREAT` служит для создания нового сегмента. Если этого флага нет, то функция `shmget()` будет искать сегмент, соответствующий ключу `key` и затем проверит, имеет ли пользователь права на доступ к сегменту.;
- 2) `IPC_EXCL` используется совместно с `IPC_CREAT` для того, чтобы не создавать существующий сегмент заново.

Системный вызов `shmctl()` изменяет управляющие параметры сегмента разделяемой памяти. Системный вызов возвращает указатель на сегмент разделяемой памяти, иначе возвращается -1. Данный системный вызов представлен в листинге 1.27.

Листинг 1.27 – Системный вызов `shmctl()`

```
1 int shmctl(int shmid, int cmd, struct shmid_ds __user *buf);
```

В качестве параметров принимаются:

- `shmid` — идентификатор сегмента разделяемой памяти;
- `cmd` — команда, которая будет выполнена над сегментом разделяемой памяти;
- `buf` — указатель на дескриптор в таблице сегментов разделяемой памяти.

Доступные команды над сегментом разделяемой памяти:

- 1) `IPC_STAT` — идентификатор сегмента разделяемой памяти;
- 2) `IPC_SET` — устанавливает значение `ipc_perm`-элемента структуры `shmid_ds`. Сами величины берет из аргумента `buf`;
- 3) `IPC_RMID` — помечает сегмент для удаления.

Команда `IPC_RMID` в действительности не удаляет сегмент из ядра, а только помечает для удаления. Настоящее же удаление не происходит, пока последний процесс, привязанный к сегменту, не отвяжется от него как следует. Конечно, если ни один процесс не привязан к сегменту на данный момент, удаление осуществляется немедленно.

Системный вызов `shmat()` привязывает процесс к сегменту разделяемой памяти. В случае успеха возвращает адрес сегмента разделяемой памяти, иначе возвращается -1. Данный системный вызов представлен в листинге 1.28.

Листинг 1.28 – Системный вызов `shmat()`

```
1 int shmat(int shmid, char __user *shmaddr, int shmflg);
```

В качестве параметров принимаются:

- `shmid` — идентификатор сегмента разделяемой памяти;
- `shmaddr` — адрес сегмента разделяемой памяти, которому необходимо привязаться, если `NULL`, то ядро пытается найти нераспределенную область. Если значение `shmaddr` не равно `NULL`, а в `shmflg` указан флаг `SHM_RND`, то подключение производится по адресу `shmaddr`, округлённому до ближайшего значения кратного `SHMLBA`. В противном случае `shmaddr` должно быть выровнено по адресу страницы, к которому производится подключение;
- `shmflg` — флаги прав доступа к сегменту.

Системный вызов `shmdt()` отвязывает процесс от сегмента разделяемой памяти по указанному адресу. Системный вызов в случае успеха возвращает 0, иначе -1. Данный системный вызов представлен в листинге 1.29.

Листинг 1.29 – Системный вызов `shmdt()`

```
1 int shmdt(char __user *shmaddr);
```

После того, как разделяемый сегмент памяти больше не нужен процессу, он должен быть отсоединен вызовом `shmdt()`. Как уже отмечалось, это не то же самое, что удаление сегмента из ядра. После успешного отсоединения значение элемента `shm_nattch` структуры `shmid_ds` уменьшается на 1. Когда оно достигает 0, ядро физически удаляет сегмент.

### 1.2.6 Программные каналы

Канал представляет собой средство связи стандартного вывода одного процесса со стандартным вводом другого. Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как другой применяется для получения данных из канала (чтение). В этом смысле, канал мало применим практически, так как создающий его процесс может использовать канал только для взаимодействия с самим собой.

Программные каналы бывают именованные и неименованные. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор. Процесс–потомок наследует все дескрипторы открытых файлов процесса–предка, в том числе и неименованных программных каналов. Программные каналы имеют встроенные средства взаимоисключения — массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают.

Системный вызов `pipe()` создает неименованный программный канал. В качестве параметра принимает массив из двух целых, которые связаны между собой. Данный системный вызов представлен в листинге 1.30.

Листинг 1.30 – Системный вызов `pipe()`

```
1 int pipe(int __user *fildes);
```

Системный вызов `pipe2()` создает неименованный программный канал, также как и `pipe()`. В качестве параметра принимает в отличии от `pipe()` флаги прав доступа. Данный системный вызов представлен в листинге 1.31.

Листинг 1.31 – Системный вызов `pipe2()`

```
1 int pipe2(int __user *fildes, int flags);
```

Системный вызов `close()` закрывает файловый дескриптор. В качестве параметра принимает файловый дескриптор на закрытие. Данный системный вызов представлен в листинге 1.32.

Листинг 1.32 – Системный вызов `close()`

```
1 int close(unsigned int fd);
```

## 1.3 Подход перехвата функций

Перехват функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции так, чтобы при вызове перехватываемой функции управление передвалось подменяемой функции. Данная функция выполняется вместо системной функции, выполняя сначала и после вызова оригинальной функции необходимые действия.

Перехватчик можно устанавливать из загружаемого GPL-модуля, без пересборки ядра. Подход поддерживает ядра версий 3.19+ для архитектуры `x86_64`.

В данной работе необходимо использовать перехват функций для взаимодействия с сигналами, семафорами, сегментами разделяемой памяти и программными каналами, по той причине что дескриптор процесса не предоставляет такой функционал.

Далее рассмотрим наиболее известные подходы перехвата функций.

### 1.3.1 `kprobes`

`kprobes` [4] – специальный интерфейс, предназначенный для отладки и трассировки ядра. Данный интерфейс позволяет устанавливать пред- и пост-обработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут изменять их значение. Таким образом, `kprobes` можно использовать как и в целях мониторинга, так и для возможности повлиять на дальнейший ход

работы ядра [5].

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре – это реализуется с помощью точек останова (инструкция `int3`), внедряемых в исполняемый код ядра. Таким образом, можно перехватить любую функцию в ядре;
- хорошо задокументированный API;
- нетривиальные накладные расходы – для расстановки и обработки точек останова необходимо большое количество процессорного времени [5];
- техническая сложность реализации. Так, например, чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте на стеке они находятся, и самостоятельно их оттуда извлекать;
- при подмене адреса возврата из функции используется стек, реализованный с помощью буфера фиксированного размера. Таким образом, при большом количестве одновременных вызовов перехваченной функции, могут быть пропущены срабатывания.

### 1.3.2 **ftrace**

**ftrace** [6] – это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора `-pg` [7] и `mfentry` [7]. Данные функции вставляют в начало каждой функции вызов специальной трассировочной функции `mcount()` или `__fentry()`. В пользовательских программах данная возможность компилятора используется профилировщиками, с целью отслеживания всех вызываемых функций. В ядре эти функции используются исключительно для реализации рассматриваемого фреймворка.

Для большинства современных архитектур процессора доступна оптимизация: динамический **ftrace** [7]. Ядро знает расположение всех вызовов функций `mcount()` или `__fentry()` и на ранних этапах загрузки ядра подменяет их машинный код на специальную машинную инструкцию `NOP` [8],

которая ничего не делает. При включении трассировки, в нужные функции необходимые вызовы добавляются обратно. Если **ftrace** не используется, его влияние на производительность системы минимально.

Особенности рассматриваемого фреймворка:

- имеется возможность перехватить любую функцию;
- перехват совместим с трассировкой;
- фреймворк зависит от конфигурации ядра, но, в популярных конфигурациях ядра (и, соответственно, в популярных образах ядра) установлены все необходимые флаги для работы;

Структура **struct ftrace\_hook** описывает каждую перехватываемую функцию. Данная структура представлена в листинге 1.33.

Листинг 1.33 – **struct ftrace\_hook**

```
1 struct ftrace_hook {  
2     const char *name;  
3     void *function;  
4     void *original;  
5  
6     unsigned long address;  
7     struct ftrace_ops ops;  
8 };
```

- **name** — имя перехватываемой функции;
- **function** — адрес функции обёртки, вызываемой вместо перехваченной функции;
- **original** — указатель на перехватываемую функцию.

Остальные поля считаются деталью реализации. Описание всех перехватываемых были собраны в массив, а для инициализации используется специальный макрос, который представн в листинге 1.34.

### Листинг 1.34 – Макрос HOOK()

```
1 #define HOOK(_name, _function, _original) \
2 { \
3     .name = SYSCALL_NAME(_name), \
4     .function = (_function), \
5     .original = (_original), \
6 }
```

## 1.4 Простанство ядра и пользователя

Для взаимодействия приложений с ядром и ядра с приложениями используются следующие функции ядра.

Функция `copy_to_user` копирует данные из ядра в пространство пользователя

### Листинг 1.35 – Функция `copy_to_user`

```
1 long copy_to_user(void __user *to, const void *from, long n);
```

В качестве параметров данная функция принимает:

- `to` — адрес назначения находится в пространстве пользователя;
- `from` — адрес источника находится в пространстве ядра;
- `n` — количество копируемых байт.

Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

Функция `copy_from_user` копирует данные из пространства пользователя в пространство ядра.

### Листинг 1.36 – Функция `copy_from_user`

```
1 long copy_from_user(void *to, const void __user *from, long n);
```

В качестве параметров данная функция принимает:

- `to` — адрес назначения находится в пространстве ядра;



- `from` — адрес источника находится в пространстве поль зователя;
- `n` — количество копируемых байт.

Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

Если какие-то данные не могут быть скопированы, эта функция. Если некоторые данные не могут быть скопированы, эта функция добавит нулевые байты к скопированным данным до требуемого размера.

## 1.5 Виртуальная файловая система `/proc`

Виртуальная файловая система `/proc` — специальный интерфейс, с помощью которого можно мгновенно получить некоторую информацию о ядре в пространство пользователя и передать информацию в пространство ядра. `/proc` отображает в виде дерева каталогов внутренние структуры ядра.

В основном дереве, каждый каталог имеет числовое имя и соответствует процессу, с соответствующим PID. Файлы в этих каталогах соответствуют структуре `task_struct`. Так, например, с помощью команды `cat /proc/1/cmdline`, можно узнать аргументы запуска процесса с идентификатором равным единице. В дереве `/proc/sys` отображаются внутренние переменные ядра.

Таким образом ядро предоставляет возможность добавить свое дерево в виртуальную файловую систему. Для этого существует специальная структура `struct proc_ops` содержащая указатели на функции взаимодействия с файлом, такие как открытие, закрытие, чтение и запись, заменяющая аналогичную структуру `struct file_operations` для файлов на диске. В листинге 1.37 представлено объявление данной структуры в ядре.

### Листинг 1.37 – struct proc\_ops

```

1 struct proc_ops {
2     unsigned int proc_flags;
3     int (*proc_open)(struct inode *, struct file *);
4     ssize_t (*proc_read)(struct file *, char __user *, size_t,
5         loff_t *);
6     ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
7     ssize_t (*proc_write)(struct file *, const char __user *,
8         size_t, loff_t *);
9     /* mandatory unless nonseekable_open() or equivalent is used
10        */
11     loff_t (*proc_lseek)(struct file *, loff_t, int);
12     int (*proc_release)(struct inode *, struct file *);
13     __poll_t (*proc_poll)(struct file *, struct poll_table_struct
14         *);
15     long (*proc_ioctl)(struct file *, unsigned int, unsigned
16         long);
17 #ifdef CONFIG_COMPAT
18     long (*proc_compat_ioctl)(struct file *, unsigned int,
19         unsigned long);
20 #endif
21     int (*proc_mmap)(struct file *, struct vm_area_struct *);
22     unsigned long (*proc_get_unmapped_area)(struct file *,
23         unsigned long, unsigned long, unsigned long, unsigned
24         long);
25 } __randomize_layout

```

Для изменения поведения файла при чтении или записи необходимо создать экземпляр структуры со своими функциями чтения и записи.

Системный вызов `proc_mkdir` создает в виртуальной файловой системе `/proc` директорию. В листинге 1.38 представлен заголовок этой функции.

### Листинг 1.38 – Функция `proc_mkdir`

```
1 struct proc_dir_entry *proc_mkdir(const char *name, struct
    proc_dir_entry *parent);
```

В качестве параметров данная функция принимает:

- `name` — имя созданной директории;
- `parent` — указатель на структуру `proc_dir_entry`, описывающую родительскую директорию, если `NULL`, то директория создается в корне.

Системный вызов возвращает указатель на структуру `proc_dir_entry` созданной директории в случае успеха и `NULL` при неудаче

Системный вызов `proc_create` создаёт в `/proc` файл. В листинге 1.39 представлен заголовок этой функции.

### Листинг 1.39 – Функция `proc_create`

```
1 struct proc_dir_entry *proc_create(const char *name, umode_t
    mode, struct proc_dir_entry *parent, const struct proc_ops
    *proc_ops);
```

В качестве параметров данная функция принимает:

- `name` — имя создаваемого файла;
- `parent` — указатель на структуру `proc_dir_entry`, описывающую родительскую директорию, если равно `NULL`, то файл создаётся в корне;
- `proc_ops` — указатель на структуру с функциями работы с файлом.

Системный вызов `proc_symlink` создаёт в `/proc` символическую ссылку. В листинге 1.40 представлен заголовок этой функции.

### Листинг 1.40 – Функция `proc_symlink`

```
1 struct proc_dir_entry *proc_symlink(const char *name, struct
    proc_dir_entry *parent, const char *dest);
```

В качестве параметров данная функция принимает:

- `name` — имя создаваемой символической ссылки;

- `parent` — указатель на структуру `proc_dir_entry`, у, описывающую родительскую директорию, если равно `NULL`, то символическая ссылка создаётся в корне;
- `dest` — имя файла, для которого создаётся символическая ссылка.

Системный вызов возвращает указатель на структуру `proc_dir_entry` созданной символической ссылки в случае успеха и `NULL` при неудаче

## Вывод

В данном разделе были проанализированы подходы к перехвату функций — `ftrace` позволяет перехватить любую функцию зная лишь её имя, может быть загружен в ядро динамически и не требует специальной сборки ядра и имеет хорошо задокументированный API. Были рассмотрены структуры и функции ядра, предоставляющие информацию о процессах и памяти, отправленных сигналов процессов, программных каналов, семафоров и сегментах разделяемой памяти; рассмотрены системные вызовы для работы с сигналами, семафорами, программными каналами и семафорами для дальнейшей их подмены с помощью фреймворка `ftrace`.

## 2 Конструкторская часть

### 2.1 Последовательность действий

На рисунке 2.1 представлена последовательность действий, выполняемых при загрузке модуля, а на рисунке 2.2 — при выгрузке модуля.

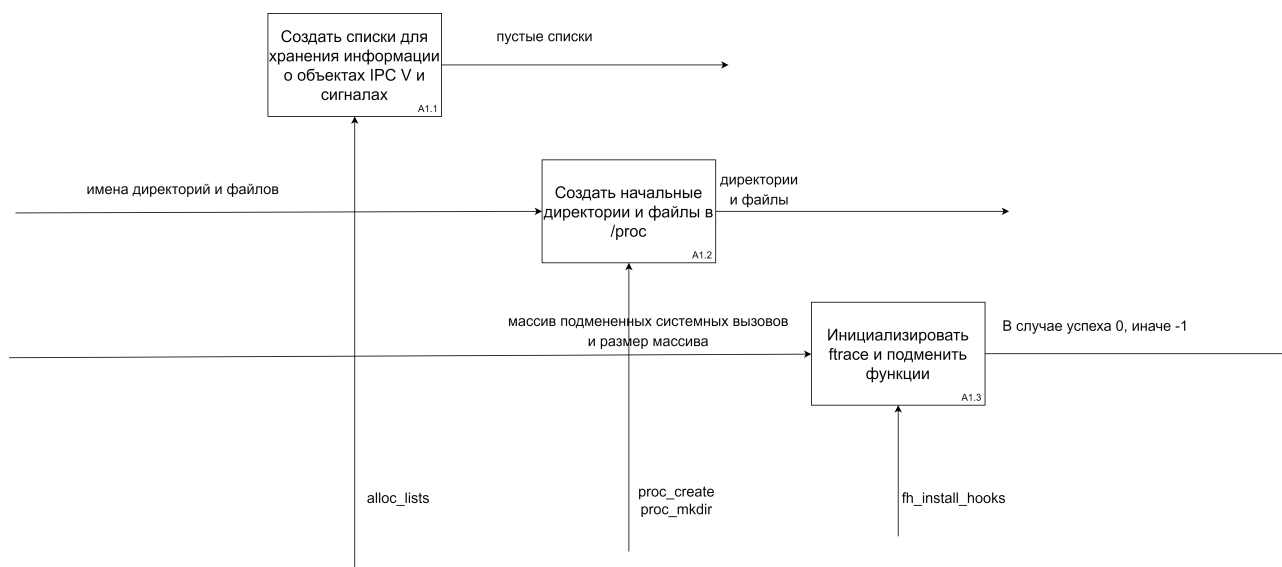


Рис. 2.1 – Последовательность действий при загрузке модуля

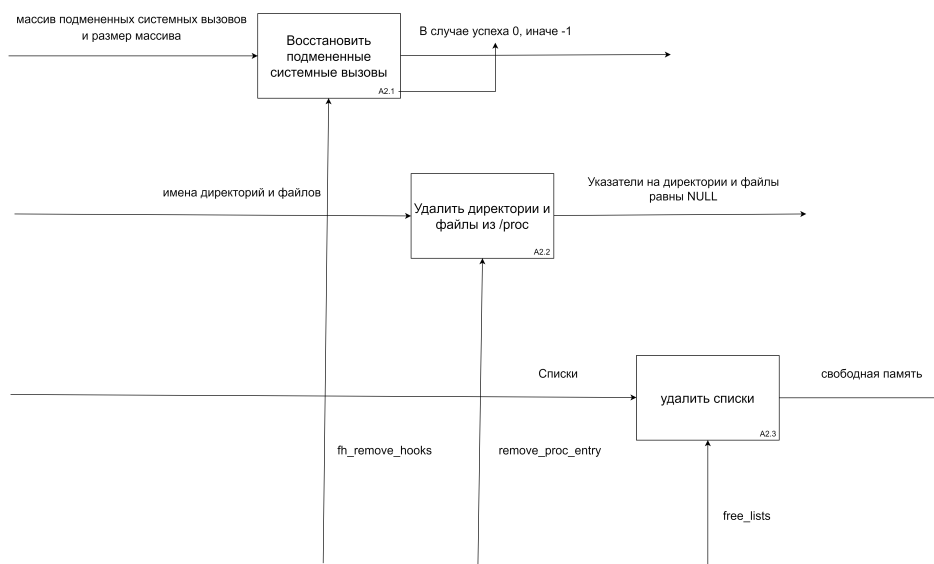


Рис. 2.2 – Последовательность действий при выгрузке модуля

## 2.2 Разработка алгоритмов

На рисунке 2.5 представлены схемы алгоритмов загрузки и выгрузки модуля.

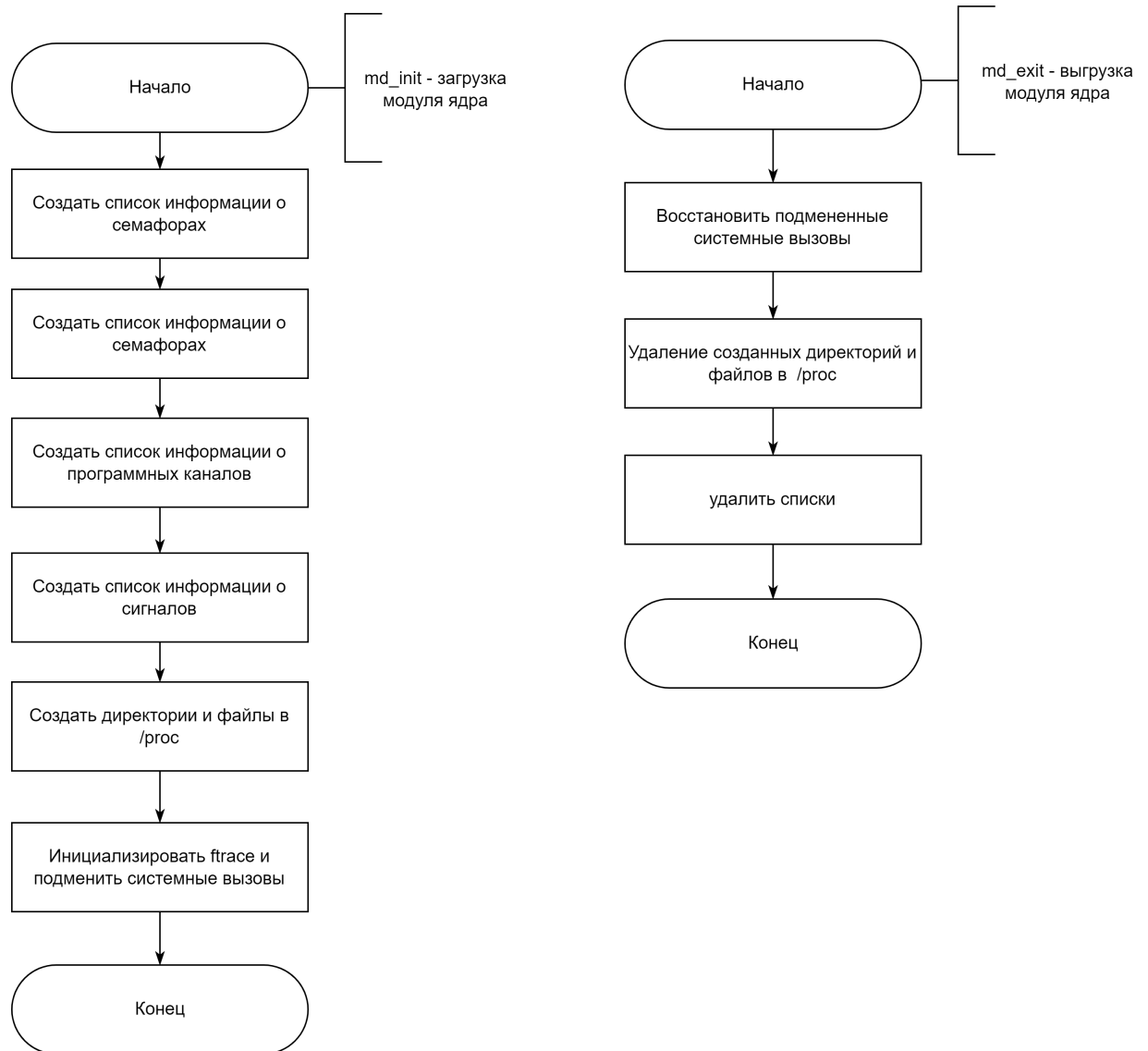


Рис. 2.3 – Схемы алгоритмов загрузки и выгрузки модуля ядра

На рисунке 2.4 представлена схемы алгоритмов создание и удаление директорий и файлов в `/proc`.

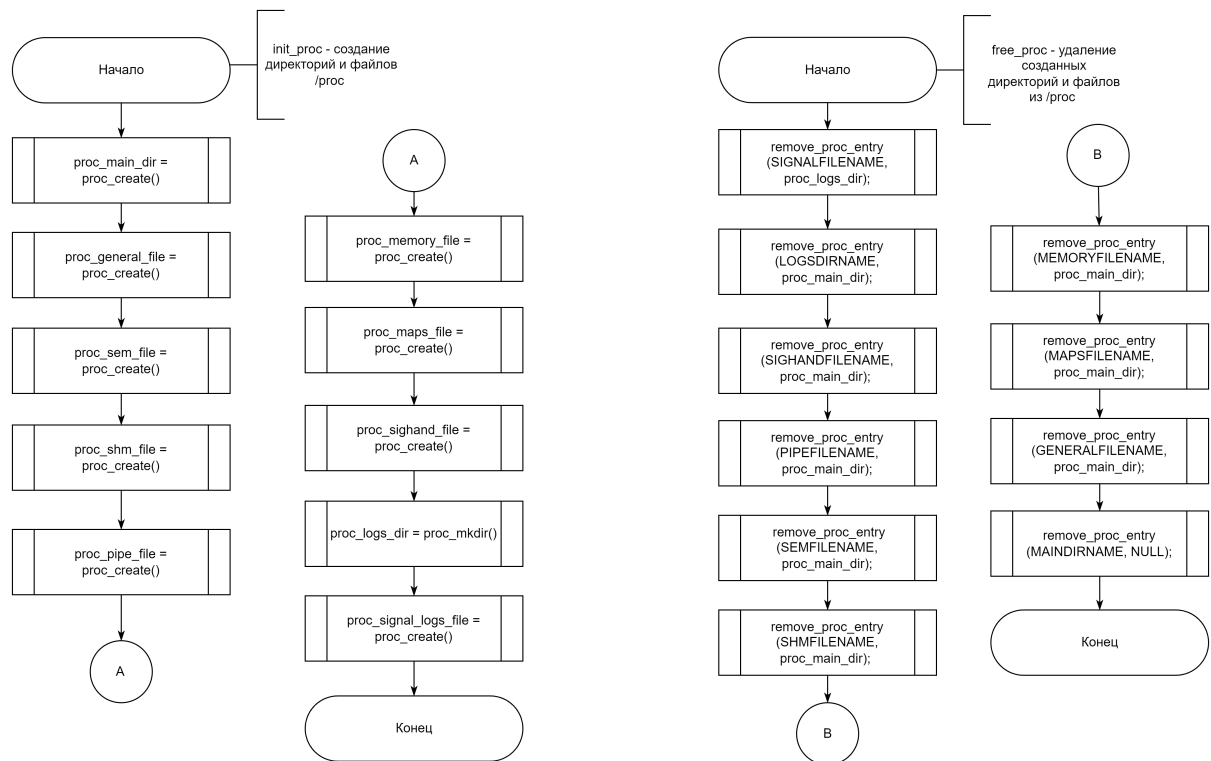


Рис. 2.4 – Схемы алгоритмов создание и удаление директорий и файлов в `/proc`

При загрузке модуля ядра в `/proc` создается директория `monitoring` с файлами:

- 1) `general` — файл, который предоставляет информацию о приоритете, времени выполнения и простоя, политики о процессах;
- 2) `memory` — файл, который предоставляет информацию о адресном пространстве процессов;
- 3) `maps` — файл, который предоставляет информацию о регионы адресного пространства процесса;
- 4) `sighands` — файл, который предоставляет информацию о назначенных обработчиках сигналов процессов. В данный файл необходимо записать `pid` процесса для получения данной информации;
- 5) `sem` — файл, который предоставляет информацию о семафорах;
- 6) `shm` — файл, который предоставляет информацию о сегментах разделя-

мой памяти;

- 7) `pipe` — файл, который предоставляет информацию о программных каналах;
- 8) `signals` — файл, который предоставляет информацию о отправленных сигналах процессами.

На рисунке 2.5 представлена схема алгоритма перехвата системных вызовов.

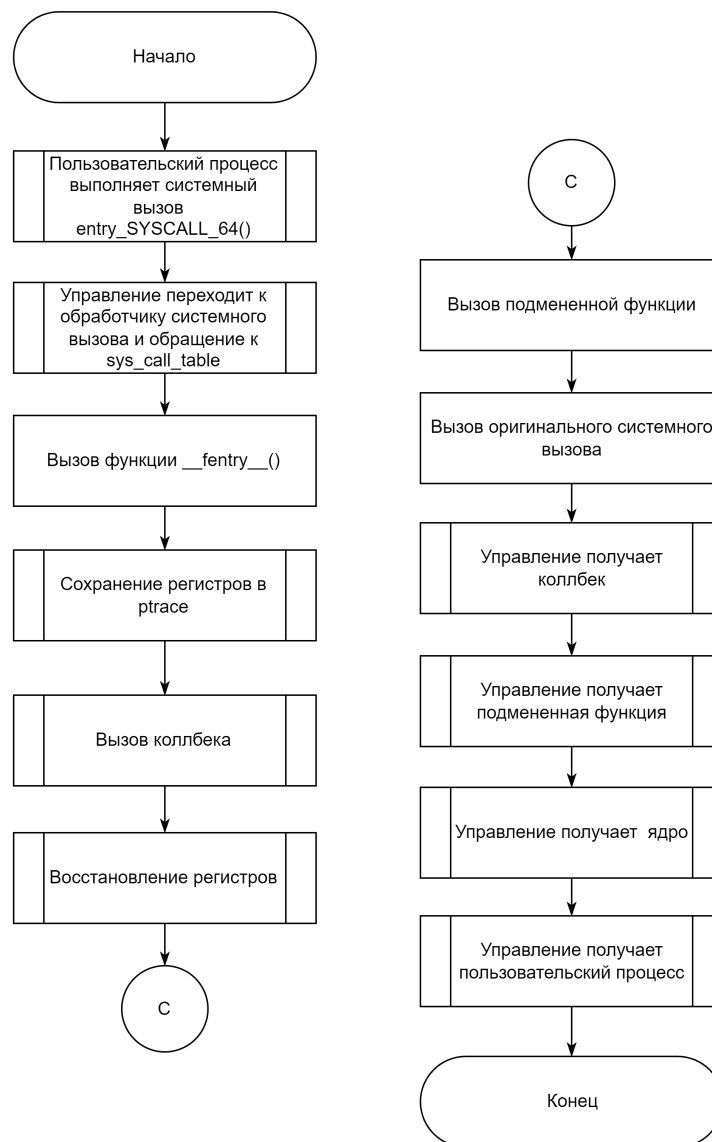


Рис. 2.5 – Схема алгоритма перехвата системного вызова

Пользовательский процесс выполняет инструкцию `SYSCALL`. С помощью этой инструкции выполняется переход в режим ядра и управление передаётся низкоуровневому обработчику системных вызовов



`entry_SYSCALL_64()`. Управление переходит к обработчику системного вызова. Ядро передаёт управление функции `do_syscall_64()`. Эта функция обращается к таблице обработчиков системных вызовов `sys_call_table` и с помощью неё вызывает конкретный обработчик системного вызова.

Благодаря безусловному переходу, управление получает наша функция `hook_sys_clone()`, а не оригинальная функция `sys_clone()`. При этом всё остальное состояние процессора и памяти остаётся без изменений — функция получает все аргументы оригинального обработчика и при завершении вернёт управление в функцию `do_syscall_64()`.

На рисунке 2.6 представлена схема алгоритма чтения из файла `general`.

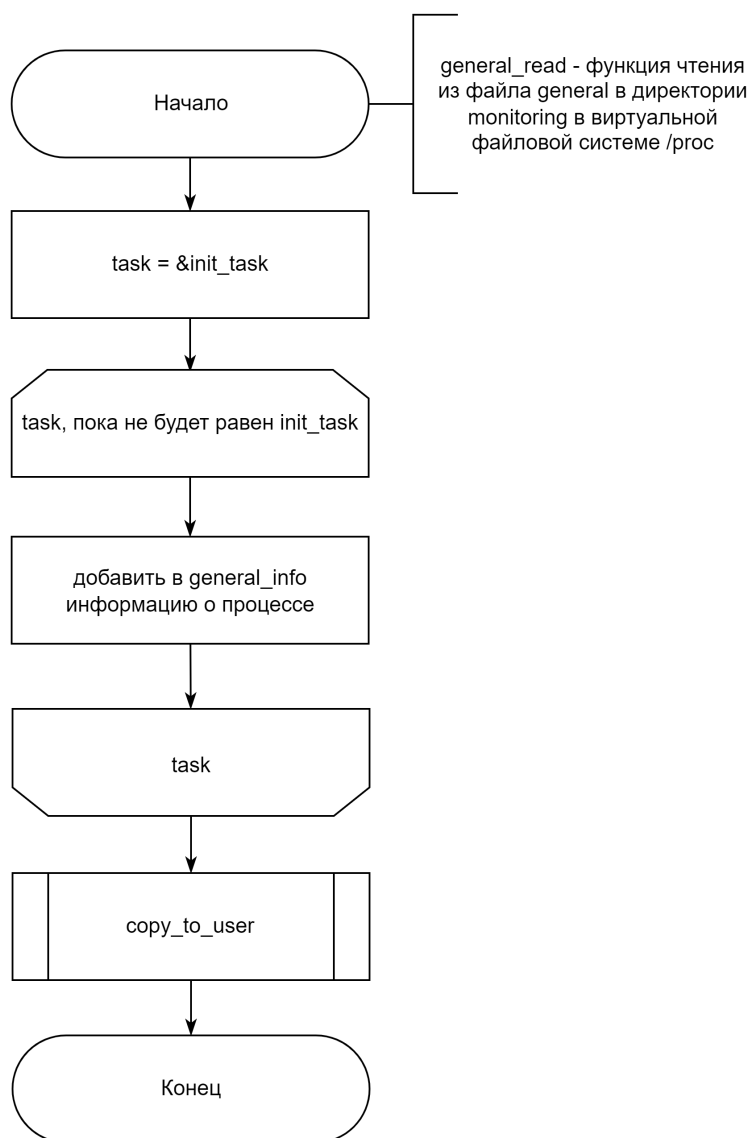


Рис. 2.6 – Схема алгоритма чтения из файла `general`

На рисунке 2.7 представлена схема алгоритма чтения из файла `sighands`.

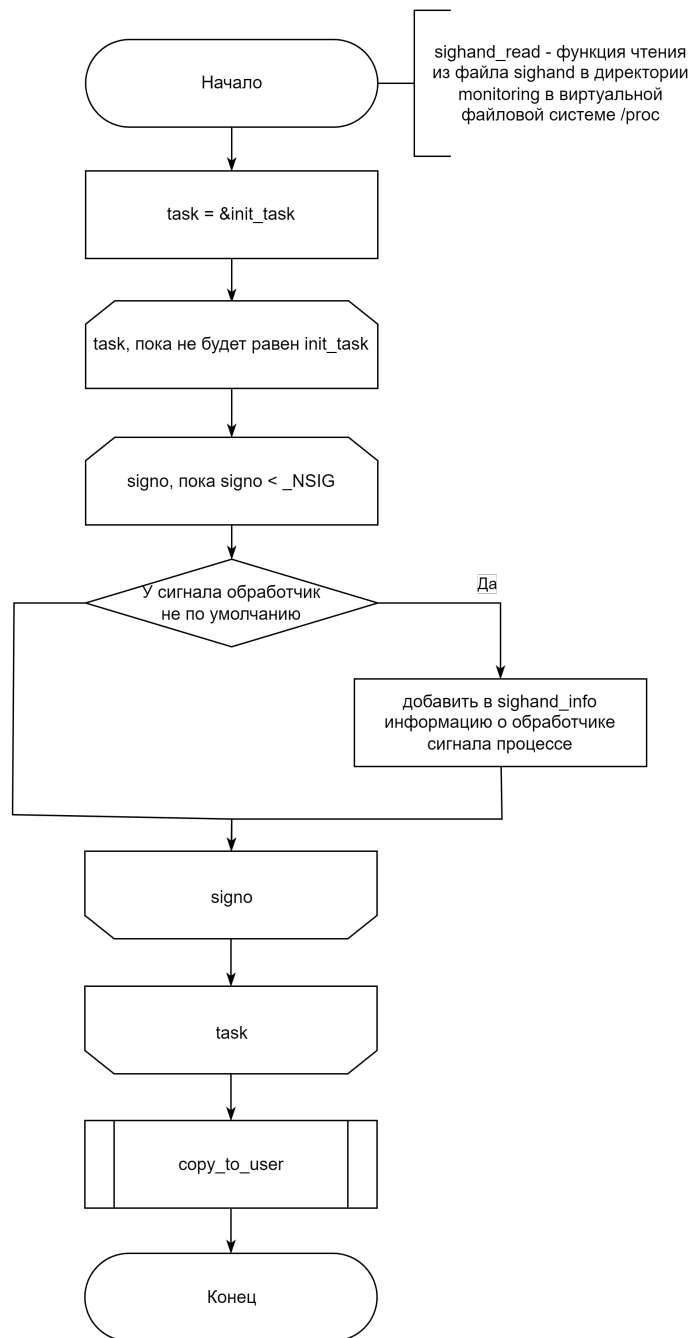


Рис. 2.7 – Схема алгоритма чтения из файла `sighands`

На рисунке 2.8 представлена схемы алгоритмов чтения из файла **shm** и **sem**.

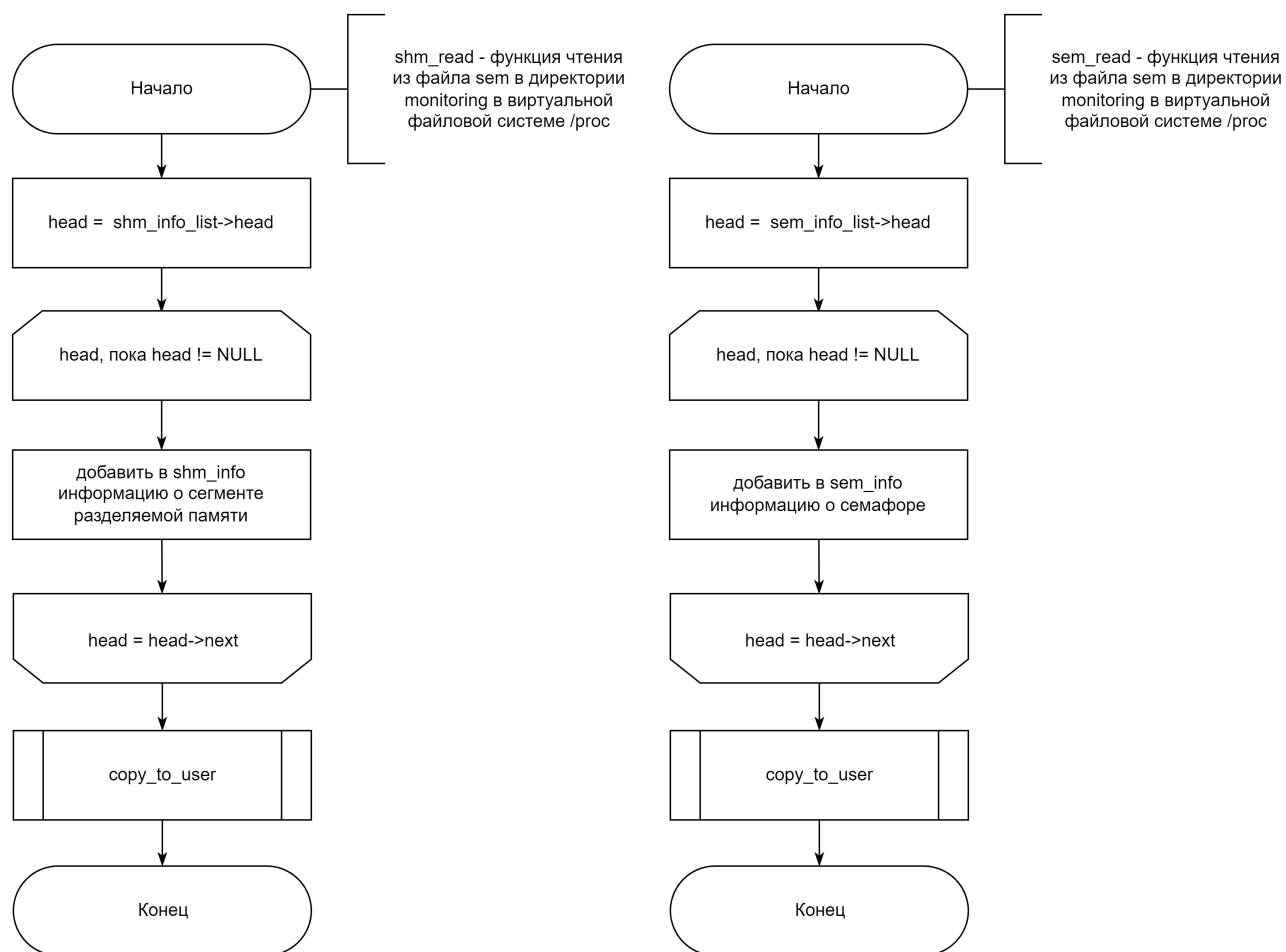


Рис. 2.8 – Схемы алгоритмов чтения из файла **shm** и **sem**

На рисунке 2.9 представлена схемы алгоритмов подменяемой функции на примере `shmget()` и `shmctl()`.

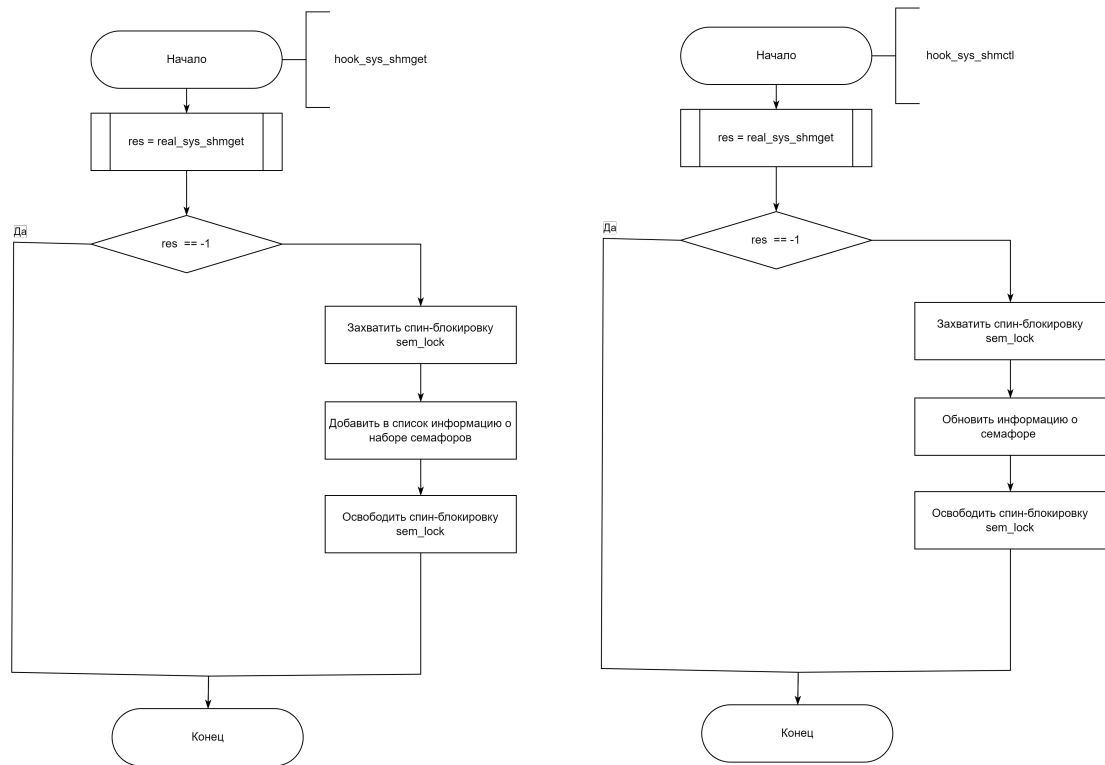


Рис. 2.9 – Схемы алгоритмов подменяемой функции на примере `shmget()` и `shmctl()`

## 3 Технологическая часть

### 3.1 Выбор языка и среды программирования

Для реализации ПО был выбран язык программирования C [9], поскольку в нём есть все инструменты для реализации загружаемого модуля ядра. Средой программирования послужил графический редактор Visual Studio Code [10], так как в нём много плагинов, улучшающих процесс разработки.

### 3.2 Реализация загружаемого модуля

В листингах 3.1–3.2 представлена функция загрузки модуля, а в листинге 3.3 функция выгрузки модуля.

Листинг 3.1 – Функция загрузки модуля

```
1 static int __init md_init(void)
2 {
3     int err;
4
5     err = alloc_lists();
6     if (err)
7     {
8         return err;
9     }
10
11     err = init_proc();
12     if (err)
13     {
14         free_proc();
15         free_lists();
16         return err;
17     }
```

### Листинг 3.2 – Функция загрузки модуля

```
1  err = install_hooks();
2  if(err)
3  {
4      printk(KERN_ERR "%s install_hooks error\n", PREFIX);
5      free_proc();
6      free_lists();
7
8      return err;
9  }
10
11  pr_info("%s: module loaded!\n", PREFIX);
12
13  return 0;
14 }
```

### Листинг 3.3 – Функция выгрузки модуля

```
1  static void __exit md_exit(void)
2  {
3      remove_hooks();
4      free_proc();
5      free_lists();
6
7      pr_info("%s: module unloaded!\n", PREFIX);
8  }
```

В листинге 3.4 представлена функция `lookup_name()`, которая возвращающей адрес функции перехватываемой функции по её названию.

Листинг 3.4 – Реализация функции `lookup_name()`

```
1 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
2 static unsigned long lookup_name(const char *name)
3 {
4     struct kprobe kp = {
5         .symbol_name = name
6     };
7     unsigned long retval;
8
9     if (register_kprobe(&kp) < 0) return 0;
10    retval = (unsigned long) kp.addr;
11    unregister_kprobe(&kp);
12    return retval;
13 }
14 #else
15 static unsigned long lookup_name(const char *name)
16 {
17     return kallsyms_lookup_name(name);
18 }
19 #endif
```

В листингах 3.5 и 3.6 представлена реализация функции, которая инициализирует структуру `ftrace_ops`.

Листинг 3.5 – Реализация функции `install_hook()`

```
1 int fh_install_hook(struct ftrace_hook *hook)
2 {
3     int err;
4
5     err = fh_resolve_hook_address(hook);
6     if (err)
7         return err;
```

Листинг 3.6 – Реализация функции `fn_install_hook()`

```

1  hook->ops.func = fh_ftrace_thunk;
2  hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
3  | FTRACE_OPS_FL_RECURSION
4  | FTRACE_OPS_FL_IPMODIFY;
5
6  err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
7  if (err) {
8      pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
9      return err;
10 }
11
12 err = register_ftrace_function(&hook->ops);
13 if (err) {
14     pr_debug("register_ftrace_function() failed: %d\n", err);
15     ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
16     return err;
17 }
18
19 return 0;
20 }
```

В листингах 3.7 и 3.8 представлена реализация отключения перехвата функции.

Листинг 3.7 – Реализация функции `fn_remove_hook()`

```

1 void fh_remove_hook(struct ftrace_hook *hook)
2 {
3     int err;
4
5     err = unregister_ftrace_function(&hook->ops);
6     if (err) {
7         pr_debug("unregister_ftrace_function() failed: %d\n",
8             err);
9     }
10 }
```



Листинг 3.8 – Реализация функции `fn_remove_hook()`

```
1   err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
2   if (err) {
3       pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
4   }
5 }
```

В листинге представлена реализация добавления перехватываемых функций в `hooks`.

Листинг 3.9 – Реализация добавления перехватываемых функций в `hooks`

```
1 static struct ftrace_hook hooks[] = {
2     HOOK("sys_kill", hook_sys_kill, &real_sys_kill),
3     HOOK("sys_signal", hook_sys_signal, &real_sys_signal),
4     HOOK("sys_semget", hook_sys_semget, &real_sys_semget),
5     HOOK("sys_semop", hook_sys_semop, &real_sys_semop),
6     HOOK("sys_semctl", hook_sys_semctl, &real_sys_semctl),
7     HOOK("sys_pipe", hook_sys_pipe, &real_sys_pipe),
8     HOOK("sys_pipe2", hook_sys_pipe2, &real_sys_pipe2),
9     HOOK("sys_close", hook_sys_close, &real_sys_close),
10    HOOK("sys_shmget", hook_sys_shmget, &real_sys_shmget),
11    HOOK("sys_shmat", hook_sys_shmat, &real_sys_shmat),
12    HOOK("sys_shmdt", hook_sys_shmdt, &real_sys_shmdt),
13    HOOK("sys_old_shmctl", hook_sys_shmctl, &real_sys_shmctl),
14    HOOK("sys_shmctl", hook_sys_shmctl, &real_sys_shmctl),
15 };
```

В листингах 3.10–3.16 представлена реализация функций оберток `sys_kill`, `sys_signal`, `sys_semget`, `sys_semop` и `sys_semctl`.

Листинг 3.10 – Реализация функции обертки `sys_kill()`

```
1 static asmlinkage long (*real_sys_kill)(const struct pt_regs *);
2 static asmlinkage int hook_sys_kill(const struct pt_regs *regs)
3 {
4     int res = real_sys_kill(regs);
5
6     if (res == 0)
7     {
8         pid_t pid = regs->di;
9         int sig = regs->si;
10
11         char currentString[TEMP_STRING_SIZE];
12
13         memset(currentString, 0, TEMP_STRING_SIZE);
14         snprintf(currentString, TEMP_STRING_SIZE, "Process %d
15             sent signal %s to process %d\n", current->pid,
16                 signal_names[sig], pid);
17
18         spin_lock(&signal_logs_lock);
19
20         strcat(signal_logs, currentString);
21
22         spin_unlock(&signal_logs_lock);
23     }
24
25     return res;
26 }
```

Листинг 3.11 – Реализация функции обертки `sys_semget()`

```

1 static void get_sem_info(int semid, int nsems, int semflg)
2 {
3     spin_lock(&sem_lock);
4     for (int semnum = 0; semnum < nsems; semnum++) {
5         sem_info_t info = {
6             .semid = semid,
7             .semnum = semnum + 1,
8             .pid = current->pid,
9             .semflg = semflg,
10            .lastcmd = -1,
11            .value = -1
12        };
13
14        push_bask_semlist(sem_info_list, info);
15    }
16    spin_unlock(&sem_lock);
17 }
18
19 static asmlinkage long (*real_sys_semget)(const struct pt_regs *);
20 static asmlinkage int hook_sys_semget(const struct pt_regs *regs)
21 {
22     int semid = real_sys_semget(regs);
23
24     key_t key = regs->di;
25     int nsems = regs->si;
26     int semflg = regs->dx;
27
28     if (semid == -1) {
29         pr_err("%s%s: Proccess %d can't create or get %d
30                semafores\n", PREFIX, SEMPREFIX, current->pid, nsems);
31     }
32     else {

```

Листинг 3.12 – Реализация функции обертки `sys_semget()`

```

1      get_sem_info(semid, nsems, semflg);
2      pr_info("%s%s: Process %d create or get %d semafores
           with semid %d\n", PREFIX, SEMPREFIX, current->pid,
           nsems, semid);
3  }
4
5      return semid;
6  }
```

Листинг 3.13 – Реализация функции обертки `sys_semop()`

```

1  static void update_semop_info(int semid, struct sembuf __user
    *sops)
2  {
3      spin_lock(&sem_lock);
4
5      semnode *head = sem_info_list->head;
6
7      for(; head; head = head->next) {
8          if (head->info.semid == semid && head->info.semnum ==
                sops->sem_num) {
9              head->info.value += sops->sem_op;
10         }
11     }
12
13     spin_unlock(&sem_lock);
14 }
15
16 static asmlinkage long (*real_sys_semop)(const struct pt_regs *);
17 static asmlinkage int hook_sys_semop(const struct pt_regs *regs)
18 {
19     int res = real_sys_semop(regs);
20
21     int semid = regs->di;
```

Листинг 3.14 – Реализация функции обертки `sys_semop()`

```

1  struct sembuf __user *sops = regs->si;
2  unsigned nsops = regs->dx;
3
4  if (res == -1) {
5      pr_err("%s%s: Proccess %d can't operate with %d semaphore
6          on semid %d\n", PREFIX, SEMPREFIX, current->pid,
7          sops->sem_num, semid);
8  }
9  else {
10     update_semop_info(semid, sops);
11     pr_info("%s%s: Proccess %d operate with %d semaphore on
12         semid %d\n", PREFIX, SEMPREFIX, current->pid,
13         sops->sem_num, semid);
14 }
15
16 return res;
17 }

```

Листинг 3.15 – Реализация функции обертки `sys_semctl()`

```

1  static void update_semctl_info(int semid, int semnum, int cmd,
2      unsigned long arg)
3  {
4      spin_lock(&sem_lock);
5      semnode *head = sem_info_list->head;
6      ushort *values = NULL;
7
8      if (cmd == SETALL || cmd == GETALL)
9          values = (ushort *) arg;
10
11     for(; head; head = head->next) {
12         if (head->info.semid == semid) {
13             if (cmd == SETVAL && head->info.semnum == semnum + 1)
14                 head->info.value = arg;
15         }
16     }
17 }

```

Листинг 3.16 – Реализация функции обертки `sys_semctl()`

```

1      else if (cmd == SETALL || cmd == GETALL)
2          head->info.value = values[head->info.semnum -
3              1];
4          head->info.lastcmd = cmd;
5      }
6  }
7  spin_unlock(&sem_lock);
8  pr_info("%s%s: Process %d semctl with %d semaphore on semid
9      %d, value: %d\n", PREFIX, SEMPREFIX, current->pid, semnum,
10      semid, arg);
11 }
12
13 static asmlinkage long (*real_sys_semctl)(const struct pt_regs *);
14 static asmlinkage int hook_sys_semctl(const struct pt_regs *regs)
15 {
16     int res = real_sys_semctl(regs);
17
18     int semid = regs->di;
19     int semnum = regs->si;
20     int cmd = regs->dx;
21     unsigned long arg = regs->r10;
22
23     if (res == 0)
24         update_semctl_info(semid, semnum, cmd, arg);
25
26     return res;
27 }

```

В листингах 3.17–3.18 представлена реализация чтения из файла `general`.

Листинг 3.17 – Реализация функции чтения из файла `general`

```
1 static ssize_t general_read(struct file *file, char __user *buf,
2                             size_t len, loff_t *fPos)
3 {
4     pr_info("%s%s: general_read called\n", PREFIX, FORTUNEPREFIX);
5     if (*fPos > 0)
6         return 0;
7
8     ssize_t strlen += sprintf(general_info + strlen, "%7s %7s %7s
9                             %7s %10s %7s %7s %7s %7s %14s %14s %14s %7s\n",
10    "PPID", "PID", "STATE", "ESTATE", "FLAGS", "POLICY", "PRIO",
11    "SPRIO", "NPRIO", "PRPRIO", "UTIME", "STIME", "DELAY",
12    "COMM");
13
14    struct task_struct *task = &init_task;
15    do {
16        strlen += sprintf(general_info + strlen, "%7d %7d %7d %7d
17                        %10x %7d %7d %7d %7d %7d %14llu %14llu %14llu\t%s\n",
18        task->parent->pid, task->pid, task->__state,
19        task->exit_state, task->flags,
20        task->policy, task->prio, task->static_prio,
21        task->normal_prio, task->rt_priority,
22        task->utime, task->stime, task->sched_info.run_delay,
23        task->comm);
24    }
25    while ((task = next_task(task)) != &init_task);
26
27    if (copy_to_user(buf, general_info, strlen)) {
28        printk(KERN_ERR "%s%s: copy_to_user error\n", PREFIX,
29                FORTUNEPREFIX);
30        return -EFAULT;
31    }
32 }
```

Листинг 3.18 – Реализация функции чтения из файла general

```
1 *fPos += strlen ;
2 memset(general_info , 0 , LOG_SIZE);
3
4 return strlen ;
5 }
```

В листингах 3.19–?? представлена реализация чтения из файла sighands.

Листинг 3.19 – Реализация функции чтения из файла sighands

```
1 static ssize_t sighand_read(struct file *file , char __user *buf ,
   size_t len , loff_t *fPos)
2 {
3     pr_info("%s%s: signal_read called\n", PREFIX, FORTUNEPREFIX);
4
5     if (*fPos > 0)
6         return 0;
7
8     ssize_t strlen = 0;
9     strlen += sprintf(sighand_info + strlen ,
10         "%7s\t%14s\t%8s\t%7s\t%7s\n", "PID", "SIGNAL", "FLAGS",
11         "HANDLER");
12
13     struct task_struct *task = &init_task;
14     do {
15         for (int signo = 1; signo < _NSIG; ++signo) {
16             struct k_sigaction *ka = &task->sighand->action[signo
17                 - 1];
18
19             if (ka->sa.sa_handler > 1) {
20                 strlen += sprintf(sighand_info + strlen , "%7d
21                     %14s %7lu 0x%x\n", task->pid ,
22                     signal_names[signo], ka->sa.sa_flags ,
23                     ka->sa.sa_handler);
24             }
25         }
26     } while (len > strlen);
```



### Листинг 3.20 – Реализация функции чтения из файла sighands

```

1      }
2      }
3  }
4  while ((task = next_task(task)) != &init_task);
5
6  if (copy_to_user(buf, sighand_info, strlen)) {
7      printk(KERN_ERR "%s: copy_to_user error\n", PREFIX);
8      return -EFAULT;
9  }
10
11  *fPos += strlen;
12
13  memset(sighand_info, 0, LOG_SIZE);
14
15  return strlen;
16 }
```

В листингах 3.21–3.22 представлена реализация чтения из файла memory.

### Листинг 3.21 – Реализация функции чтения из файла memory

```

1 static ssize_t memory_read(struct file *file, char __user *buf,
2 size_t len, loff_t *fPos)
3 {
4     pr_info("%s%s: memory_read called\n", PREFIX, FORTUNEPREFIX);
5
6     if (*fPos > 0)
7         return 0;
8
9     ssize_t strlen = 0;
10    strlen += sprintf(memory_info + strlen, "%7s %7s %10s %10s
        %10s %10s %10s %7s %10s %10s %10s %10s %10s\n",
        "PID", "MMUSERS", "TOTAL VM", "LOCKED VM", "DATA VM", "EXEC
        VM", "STACK VM", "MAPS", "HEAP", "CODE", "DATA", "ARGS",
        "ENV");
```

Листинг 3.22 – Реализация функции чтения из файла memory

```

1  struct task_struct *task = &init_task;
2  do {
3      struct mm_struct *mm = task->mm;
4
5      if (mm != NULL) {
6          unsigned long brk = mm->brk - mm->start_brk;
7          unsigned long code = mm->end_code - mm->start_code;
8          unsigned long data = mm->end_data - mm->start_data;
9          unsigned long args = mm->arg_end - mm->arg_start;
10         unsigned long env = mm->env_end - mm->env_start;
11
12         strlen += sprintf(memory_info + strlen, "%7d %7d
           %10lu %10lu %10lu %10lu %10lu %7d %10lu %10lu
           %10lu %10lu %10lu\n",
13         task->pid, mm->mm_users.counter, mm->total_vm,
           mm->locked_vm, mm->data_vm, mm->exec_vm,
           mm->stack_vm, mm->map_count, brk, code, data,
           args, env);
14     }
15 }
16 while ((task = next_task(task)) != &init_task);
17
18 if (copy_to_user(buf, memory_info, strlen)){
19     printk(KERN_ERR "%s: copy_to_user error\n", PREFIX);
20     return -EFAULT;
21 }
22
23 *fPos += strlen;
24 memset(memory_info, 0, LOG_SIZE);
25
26 return strlen;
27 }

```

В листинге 3.23 представлена реализация записи в файл maps.

Листинг 3.23 – Реализация функции записи в файл maps

```
1 static ssize_t maps_write(struct file *file, const char __user
    *ubuf, size_t len, loff_t *fPos)
2 {
3     pr_info("%s%s: maps_write called\n", PREFIX, FORTUNEPREFIX);
4
5     char kbuf[10];
6     if (copy_from_user(kbuf, ubuf, len)){
7         printk(KERN_ERR "%s%s: copy_from_user error\n", PREFIX,
            FORTUNEPREFIX);
8         return -EFAULT;
9     }
10    kbuf[len - 1] = 0;
11
12    if(sscanf(kbuf, "%d", &mt_pid) != 1)
13    {
14        printk(KERN_ERR "%s: sscanf error\n", PREFIX);
15        return -EFAULT;
16    }
17
18    return len;
19 }
```

В листингах 3.24–3.26 представлена реализация чтения из файла maps.

Листинг 3.24 – Реализация функции чтения из файла maps

```
1 static ssize_t maps_read(struct file *file, char __user *buf,
    size_t len, loff_t *fPos)
2 {
3     pr_info("%s%s: maps_read called\n", PREFIX, FORTUNEPREFIX);
4
5     if (*fPos > 0)
6         return 0;
7
8     ssize_t strlen = 0;
```

Листинг 3.25 – Реализация функции чтения из файла maps

```

1  struct task_struct *task = find_task_struct(mt_pid);
2
3  if (task == NULL || mt_pid == -1)
4  {
5      strlen += sprintf(maps_info + strlen, "Process with pid
6          %d doesn't exist\n", mt_pid);
7  }
8  else
9  {
10     strlen += sprintf(maps_info + strlen, "%7s %15s %10s %10s
11         %10s\n", "PID", "addr-addr", "Flags", "BYTES",
12         "PAGES");
13
14     struct mm_struct *mm = task->mm;
15
16     if (mm == NULL)
17         strlen += sprintf(maps_info + strlen, "%7d %20s %10s
18             %10s %10s\n", task->pid, "?-?", "?", "?", "?");
19     else{
20         struct vm_area_struct *vma = mm->mmap;
21
22         if (vma == NULL)
23             strlen += sprintf(maps_info + strlen, "%7d %15s
24                 %10s %10s %10s\n", task->pid, "?-?", "?", "?",
25                 "?");
26         else {
27             for (; vma != NULL; vma = vma->vm_next){
28                 unsigned long bytes = vma->vm_end -
29                     vma->vm_start;
30                 int pages = bytes / 4096;
31
32                 strlen += sprintf(maps_info + strlen, "%7d
33                     %x-%x %10lld %10lu %7d\n",

```

Листинг 3.26 – Реализация функции чтения из файла maps

```

1      task->pid , vma->vm_start , vma->vm_end ,
        vma->vm_flags , bytes , pages);
2
3      }
4
5  }
6
7  if (copy_to_user(buf, maps_info, strlen)) {
8      printk(KERN_ERR "%s%s: copy_to_user error\n", PREFIX,
9              FORTUNEPREFIX);
10     return -EFAULT;
11 }
12
13 memset(maps_info, 0, LOG_SIZE);
14
15 *fPos += strlen;
16
17 return strlen;
18 }

```

В листингах 3.27–3.28 представлена реализация чтения из файла pipe.

Листинг 3.27 – Реализация функции чтения из файла pipe

```

1 static ssize_t pipe_read(struct file *file, char __user *buf,
2     size_t len, loff_t *fPos)
3 {
4     pr_info("%s%s: pipe_read called\n", PREFIX, FORTUNEPREFIX);
5
6     if (*fPos > 0)
7         return 0;
8
9     ssize_t strlen = 0;
10    strlen += sprintf(pipes_info + strlen, "%7s %18s %7s %s\n",
11        "PID", "FD", "COUNT", "PIDS");

```

Листинг 3.28 – Реализация функции чтения из файла pipe

```

1  pnode *head = pipe_info_list.head;
2
3  for (; head; head = head->next) {
4      int count = 0;
5      ssize_t clen = 0;
6      char temp[TEMP_STRING_SIZE] = { 0 };
7
8      list_head *pos;
9      task_struct *task, *child;
10     task = pid_task(find_vpid(head->pid), PIDTYPE_PID);
11
12     list_for_each(pos; task->children; list) {
13         child = list_entry(pos; struct task_struct, sibling);
14         clen += sprintf(temp + clen, "%d,", child->pid);
15         count++;
16     }
17
18     strlen += sprintf(pipes_info + strlen, "%7d %18llu %7d
19         %s\n",
20     head->ppid, head->fd, count, temp);
21 }
22
23 if (copy_to_user(buf, pipes_info, strlen)) {
24     printk(KERN_ERR "%s%s: copy_to_user error\n", PREFIX,
25         FORTUNEPREFIX);
26     return -EFAULT;
27 }
28
29 *fPos += strlen;
30 memset(pipes_info, 0, LOG_SIZE);
31 return strlen;
32 }

```

В листинге 3.29 представлена реализация чтения из файла sem.

Листинг 3.29 – Реализация функции чтения из файла sem

```
1 static ssize_t sem_read(struct file *file, char __user *buf,  
    size_t len, loff_t *fPos)  
2 {  
3     pr_info("%s%s: sem_read called\n", PREFIX, FORTUNEPREFIX);  
4  
5     if (*fPos > 0)  
6         return 0;  
7  
8     ssize_t strlen = 0;  
9     strlen += sprintf(sem_info + strlen, "%7s %7s %7s %7s %7s  
    %7s\n", "PID", "SEMID", "SEMNUM", "FLAGS", "CMD", "VALUE");  
10    semnode *head = sem_info_list->head;  
11    for (; head; head = head->next) {  
12        char command[10] = { 0 };  
13        cmd_to_str(command, head->info.lastcmd);  
14  
15        strlen += sprintf(sem_info + strlen, "%7d %7d %7d %7d %7s  
            %7d\n",  
16            head->info.pid, head->info.semid, head->info.semnum,  
            head->info.semflg, command, head->info.value);  
17    }  
18  
19    if (copy_to_user(buf, sem_info, strlen)) {  
20        printk(KERN_ERR "%s%s: copy_to_user error\n", PREFIX,  
            FORTUNEPREFIX);  
21        return -EFAULT;  
22    }  
23  
24    *fPos += strlen;  
25    memset(sem_info, 0, LOG_SIZE);  
26    return strlen;  
27 }
```

В листингах 3.30–3.31 представлена реализация чтения из файла pipe.

Листинг 3.30 – Реализация функции чтения из файла shm

```
1 static ssize_t shm_read(struct file *file, char __user *buf,
2   size_t len, loff_t *fPos)
3 {
4     pr_info("%s%s: shm_read called\n", PREFIX, FORTUNEPREFIX);
5
6     if (*fPos > 0)
7         return 0;
8
9     ssize_t strlen = 0;
10    strlen += sprintf(shm_info + strlen, "%7s %7s %10s %14s
11      %7s\n", "PID", "SHMID", "CMD", "SIZE", "ADDR");
12    shmnode *head = shm_info_list->head;
13    for (; head; head = head->next) {
14        char command[10] = { 0 };
15        cmd_to_str(command, head->info.lastcmd);
16
17        if (head->info.addr == NULL) {
18            strlen += sprintf(shm_info + strlen, "%7d %7d %10s
19              %14llu %s\n",
20                head->info.pid, head->info.shmid, command,
21                head->info.size, "?");
22        }
23        else {
24            strlen += sprintf(shm_info + strlen, "%7d %7d %10s
25              %14llu 0x%p\n",
26                head->info.pid, head->info.shmid, command,
27                head->info.size, head->info.addr);
28        }
29    }
30    if (copy_to_user(buf, shm_info, strlen)) {
31        printk(KERN_ERR "%s%s: copy_to_user error\n", PREFIX,
32          FORTUNEPREFIX);
33    }
```



Листинг 3.31 – Реализация функции чтения из файла shm

```
1      return -EFAULT;
2  }
3
4      *fPos += strlen;
5      memset(shm_info, 0, LOG_SIZE);
6      return strlen;
7  }
```

Для файлов были созданы экземпляры структуры `proc_ops`, они представлены в листингах 3.32–3.33.

Листинг 3.32 – Экземпляры структуры `proc_ops`

```
1  static struct proc_ops signal_logs_ops = {
2      .proc_open = signal_logs_open,
3      .proc_read = signal_logs_read,
4      .proc_write = signal_logs_write,
5      .proc_release = signal_logs_release,
6  };
7  static struct proc_ops sighand_ops = {
8      .proc_open = sighand_open,
9      .proc_read = sighand_read,
10     .proc_write = sighand_write,
11     .proc_release = sighand_release,
12 };
13 static struct proc_ops memory_ops = {
14     .proc_open = memory_open,
15     .proc_read = memory_read,
16     .proc_write = memory_write,
17     .proc_release = memory_release,
18 };
19 static struct proc_ops maps_ops = {
20     .proc_open = maps_open,
21     .proc_read = maps_read,
22     .proc_write = maps_write,
```

### Листинг 3.33 – Экземпляры структуры proc\_ops

```
1     .proc_release = maps_release ,
2 };
3 static struct proc_ops general_ops = {
4     .proc_open = general_open ,
5     .proc_read = general_read ,
6     .proc_write = general_write ,
7     .proc_release = general_release ,
8 };
9 static struct proc_ops pipe_ops = {
10     .proc_open = pipe_open ,
11     .proc_read = pipe_read ,
12     .proc_write = pipe_write ,
13     .proc_release = pipe_release ,
14 };
15 static struct proc_ops sem_ops = {
16     .proc_open = sem_open ,
17     .proc_read = sem_read ,
18     .proc_write = sem_write ,
19     .proc_release = sem_release ,
20 };
21 static struct proc_ops shm_ops = {
22     .proc_open = shm_open ,
23     .proc_read = shm_read ,
24     .proc_write = shm_write ,
25     .proc_release = shm_release ,
26 };
```

Весь код программы представлен в Приложении А.

## 4 Исследовательский раздел

### 4.1 Технические характеристики

Технические характеристики устройства, на котором запускалась программа:

- 1) операционная система Ubuntu, 22.04.4 [11] с версией ядра 5.19.17;
- 2) память 16 Гайт;
- 3) процессор 2,5 ГГц 4-дерный процессор Intel Core i5-10300H [12].

### 4.2 Исследование работы программы

Для исследования работы были разработаны вспомогательные программы.

- 1) программа, которая создает процесс-потомок и посылает ему сигнал;
- 2) программа, в которой предок и потомок обмениваются сообщениями через программный канал;
- 3) программа на основе задачи «производство–потребление»;
- 4) программа на основе задачи «читатели–писатели».

Весь код вспомогательных программ представлен в Приложении А.

На рисунке 4.1 продемонстрирована работа загружаемого модуля при чтении из файла `general`.

PPID	PID	STATE	ESTATE	FLAGS	POLICY	PRIORITY	SPRIORITY	NPRIORITY	PRPRIORITY	UTIME	STIME	DELAY	COMM
0	0	0	0	4200002	0	120	120	120	0	0	48000000	0	swapper/0
0	1	1	0	400100	0	120	120	120	0	872000000	1048000000	208984918	systemd
0	2	1	0	208040	0	120	120	120	0	0	4000000	166568	kthreadd
2	3	1026	0	4208060	0	100	100	100	0	0	0	0	rcu_gp
2	4	1026	0	4208060	0	100	100	100	0	0	0	0	rcu_par_gp
2	5	1026	0	4208060	0	100	100	100	0	0	0	0	slub_flushwq
2	6	1026	0	4208060	0	100	100	100	0	0	0	0	netns
2	8	1026	0	4208060	0	100	100	100	0	0	0	2282	kworker/0:0H
2	10	1026	0	4208060	0	100	100	100	0	0	0	0	mm_percpu_wq
2	11	1026	0	208040	0	120	120	120	0	0	0	69506	rcu_tasks_kthre
2	12	1026	0	208040	0	120	120	120	0	0	0	2334	rcu_tasks_rude
2	13	1026	0	208040	0	120	120	120	0	0	0	0	rcu_tasks_trace
2	14	1	0	4208040	0	120	120	120	0	0	8000000	144755036	ksoftirqd/0
2	15	1026	0	208040	0	120	120	120	0	0	0	0	rcu_preempt
2	16	1	0	4208040	1	0	120	0	99	0	4000000	392132719	migration/0
2	17	1	0	4208040	1	49	120	49	50	0	0	0	idle_inject/0
2	19	1	0	4208040	0	120	120	120	0	0	4000000	22250177	cpuhp/0
2	20	1	0	4208140	0	120	120	120	0	0	0	25875213	cpuhp/1
2	21	1	0	4208040	1	49	120	49	50	0	0	0	idle_inject/1
2	22	1	0	4208040	1	0	120	0	99	0	0	619158	migration/1
2	23	1	0	4208040	0	120	120	120	0	0	12000000	147030233	ksoftirqd/1
2	25	1026	0	4208060	0	100	100	100	0	0	0	0	kworker/1:0H
2	26	1	0	4208140	0	120	120	120	0	0	0	1503485	cpuhp/2

Рис. 4.1 – Демонстрация работы программы при чтении из файла `general`

На рисунке 4.2 продемонстрирована работа загружаемого модуля при чтении из файла `memory`.

```
vladislav@vladislav:~$ cat /proc/monitoring/memory
```

PID	MMUSERS	TOTAL VM	LOCKED VM	DATA VM	EXEC VM	STACK VM	MAPS	HEAP	CODE	DATA	ARGS	ENV
1	1	41746	0	5027	2487	33	154	3039232	914829	320140	18	202
342	1	47031	0	8601	2277	33	169	1323008	93249	5444	30	656
372	1	6732	0	534	1608	33	81	2007040	643821	140952	27	587
580	1	3708	0	152	2260	33	140	135168	21981	2528	26	542
581	1	6450	0	1004	3144	33	194	3649536	315153	14544	30	589
583	2	22346	0	2203	2386	33	154	135168	22825	3352	31	665
683	3	60108	0	6615	2007	33	157	540672	81365	14352	29	527
684	1	704	0	57	455	33	24	135168	20857	3792	16	389
687	1	1929	0	147	1251	33	96	413696	80329	4928	26	509
688	1	2649	0	111	1664	33	98	245760	719029	55560	30	446
690	1	2376	0	77	609	33	54	135168	25933	1944	21	412
692	1	2801	0	824	1377	33	111	1716224	140365	6480	97	496
693	3	65559	0	6842	4645	33	389	1671168	2275405	83192	37	412
702	2	20701	0	2186	1433	33	96	241664	37765	1936	34	446
706	1	10260	0	2684	3656	33	307	2871296	2814717	283976	69	494
707	3	60879	0	7144	2083	33	181	2912256	59125	3488	32	412
708	3	60115	0	6414	2025	33	168	425984	25877	2184	35	459

Рис. 4.2 – Демонстрация работы программы при чтении из файла `memory`

На рисунке 4.3 продемонстрирована работа загружаемого модуля при записи в файл идентификатора процесса и чтении из файла `maps`.

```
vladislav@vladislav:~$ echo "706" > /proc/monitoring/maps
vladislav@vladislav:~$ cat /proc/monitoring/maps
```

PID	addr-addr	Flags	BYTES	PAGES
706	7440d000-7447a000	134217841	446464	109
706	7447a000-7472a000	134217845	2818048	688
706	7472a000-74969000	134217841	2355200	575
706	74969000-74970000	135266417	28672	7
706	74970000-749af000	135266419	258048	63
706	749af000-749f5000	135266419	286720	70
706	75254000-75511000	135266419	2871296	701
706	cddf2000-cddf4000	134217841	8192	2
706	cddf4000-cde01000	134217845	53248	13
706	cde01000-cde03000	134217841	8192	2
706	cde03000-cde04000	135266417	4096	1
706	cde04000-cde05000	135266419	4096	1
706	cde0c000-cde0f000	134217841	12288	3
706	cde0f000-cde13000	134217845	16384	4
706	cde13000-cde16000	134217841	12288	3
706	cde16000-cde17000	135266417	4096	1
706	cde17000-cde18000	135266419	4096	1
706	cde18000-cdf18000	135266419	1048576	256
706	cdf18000-cdf1c000	134217841	16384	4
706	cdf1c000-cdf32000	134217845	90112	22

Рис. 4.3 – Демонстрация работы программы при записи в файл и чтении из файла `maps`

На рисунке 4.4 продемонстрирована работа загружаемого модуля при чтении из файла `sighands`.

```
vladislav@vladislav:~$ cat /proc/monitoring/sighands
```

PID	SIGNAL	FLAGS	HANDLER
1	SIGQUIT	1140850688	0xcf817960
1	SIGILL	1140850688	0xcf817960
1	SIGABRT	1140850688	0xcf817960
1	SIGBUS	1140850688	0xcf817960
1	SIGFPE	1140850688	0xcf817960
1	SIGSEGV	1140850688	0xcf817960
1	33	469762052	0xf7c91870
342	SIGBUS	67108868	0xe6d494c0
342	33	469762052	0xe6891870
583	33	469762052	0x71491870
683	SIGINT	335544321	0xf800acc0
683	SIGTERM	335544321	0xf800acc0
683	33	469762052	0xf7c91870
684	SIGHUP	335544320	0x170b9ec0
684	SIGINT	335544320	0x170b9f80

Рис. 4.4 – Демонстрация работы программы при записи в файл и чтении из файла `sighands`

На рисунке 4.5 представлен запуск вспомогательной программы 1, в результате которой был отправлен сигнал. На рисунке 4.6 представлено чтение из файла `signals`.

```
vladislav@vladislav:~/os-cp/src/programs$ ./a.out
Parent: Sending SIGUSR1 signal to child...
PID: 5284 , PPID: 5283
Child: Waiting for signal from parent...
Received SIGUSR1 signal
Child: Signal received. Exiting.
Parent: Child process has exited. Exiting.
```

Рис. 4.5 – Демонстрация работы вспомогательной работы 1

```
vladislav@vladislav:~$ cat /proc/monitoring/logs/signals
Process 397 sent signal SIGTERM to process 5131
vladislav@vladislav:~$ cat /proc/monitoring/logs/signals
Process 397 sent signal SIGTERM to process 5131
Process 5283 sent signal SIGUSR1 to process 5284
vladislav@vladislav:~$ █
```

Рис. 4.6 – Демонстрация работы программы чтения из файла `signals`



На рисунке 4.7 представлен запуск вспомогательной программы 2, в результате два процесса обмениваются сообщениями через программный канал. На рисунке 4.8 представлено чтение из файла pipe.

```
vladislav@vladislav:~/os-cp/src/programs$ ./a.out
Parent process: PID = 5155, GROUP = 5155, child's ID = 5156
Message #1 sent to parent!
Child #1 has finished: PID = 5156
Child #1 exited with code 0
Received messages: xyz!

Parent process: PID = 5155, GROUP = 5155, child's ID = 5169
Message #2 sent to parent!
Child #2 has finished: PID = 5169
Child #2 exited with code 0
Received messages: xx, yy, zz
```

Рис. 4.7 – Демонстрация работы вспомогательной работы 2

```
vladislav@vladislav:~$ cat /proc/monitoring/pipe
```

PID	FD	COUNT
3230	140726540876704	0
3230	94129901292488	0
3182	140722403551512	2 3230,4459,
5112	94298044193736	1 5155,
5155	140733695511680	1 5156,
4412	140730428135408	1 5168,
4412	94917476854728	1 5168,

Рис. 4.8 – Демонстрация работы программы чтения из файла pipe

На рисунке 4.9–4.10 представлен запуск вспомогательной программы 2 и 3, а на рисунке 4.11 представлено чтение из файлов sem и shm.

```
vladislav@vladislav:~/os-cp/src/programs/producer_consumer$ ./a.out
Producer 6372 - a
Consumer 6373 - a
Producer 6372 - b
Consumer 6373 - b
Producer 6372 - c
Consumer 6373 - c
Producer 6372 - d
Consumer 6373 - d
Producer 6372 - e
Consumer 6373 - e
Producer 6372 - f
Consumer 6373 - f
Producer 6372 - g
```

Рис. 4.9 – Демонстрация работы вспомогательной работы 2

```

vladislav@vladislav:~/os-cp/src/programs/writer_reader$ ./a.out
Reader got value = '0'
Writer incremented = '1'
Reader got value = '1'
Reader got value = '1'
Reader got value = '1'
Reader got value = '1'
Reader got value = '1'
Writer incremented = '2'
Reader got value = '2'
Reader got value = '2'
Reader got value = '2'
Reader got value = '2'
Reader got value = '2'
Writer incremented = '3'
Writer incremented = '4'
Reader got value = '4'
Reader got value = '4'
Reader got value = '4'

```

Рис. 4.10 – Демонстрация работы вспомогательной работы 3

```

vladislav@vladislav:~$ cat /proc/monitoring/sem
  PID  SEMID  SEMNUM  FLAGS  CMD  VALUE
6372    0      1     932  SETVAL    1
6372    0      2     932  SETVAL   26
6372    0      3     932  SETVAL    0
6388    3      1     932  SETVAL    0
6388    3      2     932  SETVAL    0
6388    3      3     932  SETVAL    0
6388    3      4     932  SETVAL    0
6388    3      5     932  SETVAL    1
vladislav@vladislav:~$ cat /proc/monitoring/shm
  PID  SHMID  CMD  SIZE  ADDR
6372   37    ?    29  0x00000000fe1a799f
6373   37    ?    29  0x000000001f657079
6388   44    ?     4  0x000000001bd264b8
vladislav@vladislav:~$

```

Рис. 4.11 – Демонстрация работы программы чтения из файлов `sem` и `shm`

```

vladislav@vladislav:~$ cat /proc/monitoring/sem
  PID  SEMID  SEMNUM  FLAGS  CMD  VALUE
6372    0      1     932  SETVAL    1
6372    0      2     932  SETVAL   26
6372    0      3     932  SETVAL    0
6388    3      1     932  IPC_RMID    0
6388    3      2     932  IPC_RMID    0
6388    3      3     932  IPC_RMID    0
6388    3      4     932  IPC_RMID    0
6388    3      5     932  IPC_RMID    1
vladislav@vladislav:~$ cat /proc/monitoring/shm
  PID  SHMID  CMD  SIZE  ADDR
6372   37    ?    29  0x00000000fe1a799f
6373   37    ?    29  0x000000001f657079
6388   44  IPC_RMID    4  ?
vladislav@vladislav:~$ █

```

Рис. 4.12 – Демонстрация работы программы чтения из файлов `sem` и `shm`

# ЗАКЛЮЧЕНИЕ

Цель, поставленная в начале, была достигнута: разработан загружаемый модуль ядра для получения информации о приоритете, времени выполнении и простоя, виртуальной памяти, посланных сигналов, семафоров, сегментах разделяемой памяти и программных каналов процесса. В ходе выполнения курсовой работы были решены следующие задачи:

- проведен анализ структур и функций, предоставляющих возможность реализовать поставленную задачу;
- разработаны алгоритмы и структуру загружаемого модуля ядра, обеспечивающего отслеживание процессов.



# Список использованных источников

1. Desktop Operating System Market Share Worldwide [Электронный ресурс].  
— Режим доступа: <https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201412-202401> (дата обращения: 12.02.2024).
2. Mobile Operating System Market Share Worldwide [Электронный ресурс].  
— Режим доступа: <https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201412-202401> (дата обращения: 12.02.2024).
3. Linux source code: struct task\_struct [Электронный ресурс]. — Режим доступа: <https://elixir.bootlin.com/linux/v5.19.17/source/include/linux/sched.h#L726> (дата обращения: 12.02.2024).
4. Kernel Probes (Kprobes) [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/kprobes.html> (дата обращения: 12.02.2024).
5. Механизмы профилирования Linux – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/metrotek/blog/261003/> (дата обращения: 12.02.2024).
6. Using ftrace | Android Open Source Project [Электронный ресурс]. Режим доступа: <https://source.android.com/devices/tech/debug/ftrace> (дата обращения: 12.02.2024).
7. Трассировка ядра с ftrace – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/selectel/blog/280322/> (дата обращения: 12.02.2024).

8. NOP: No Operation (x86 Instruction Set Reference) [Электронный ресурс]. Режим доступа: [https://c9x.me/x86/html/file\\_module\\_x86\\_id\\_217.html](https://c9x.me/x86/html/file_module_x86_id_217.html) (дата обращения: 12.02.2024).
9. C language documentation [Электронный ресурс]. Режим доступа: <https://learn.microsoft.com/en-us/cpp/c-language/?view=msvc-170> (дата обращения: 12.02.2024).
10. Visual Studio Code [Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/> (дата обращения: 15.01.2024).
11. Enterprise Open Source and Linux | Ubuntu [Электронный ресурс]. Режим доступа: <https://ubuntu.com/> (дата обращения: 12.02.2024).
12. Процессор Intel® Core™ i5 [Электронный ресурс]. Режим доступа: <https://www.intel.com/processors/core/i5/docs> (дата обращения: 12.02.2024).