



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:

«Разработка загружаемого модуля ядра Linux для мониторинга
загруженности оперативной памяти и количества системных
ВЫЗОВОВ»

Студент ИУ7-76Б
(Группа)

(Подпись, дата)

А. А. Петрова
(И.О. Фамилия)

Руководитель

(Подпись, дата)

Н. Ю. Рязанова
(И.О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

Введение	4
1 Аналитическая часть	5
1.1 Постановка задачи	5
1.2 Информация об оперативной памяти	6
1.2.1 Файл /proc/meminfo	6
1.2.2 Структура struct sysinfo	6
1.2.3 Сравнительный анализ методов	7
1.3 Перехват функций	8
1.3.1 Модификация таблицы системных вызовов	8
1.3.2 kprobes	8
1.3.3 Kernel tracepoints	9
1.3.4 ftrace	9
1.3.5 Сравнительный анализ методов	10
2 Конструкторская часть	11
2.1 Структура ftrace_ops и функция коллбека	11
2.2 Алгоритм перехвата системного вызова	12
2.3 Алгоритм подсчёта количества системных вызовов	14
2.4 Алгоритм получения информации об объеме памяти	15
2.5 Точки входа	16
3 Технологическая часть	17
3.1 Выбор языка программирования	17
3.2 Информация о памяти в системе	17
3.3 Поиск адреса перехватываемой функции	18
3.4 Инициализация ftrace	20
3.5 Функции обёртки	21

3.6	Получение информации о количестве системных вызовов	22
3.7	Детали реализации	24
4	Исследовательская часть	27
4.1	Результаты работы разработанного ПО	27
4.2	Анализ результатов	29
	Заключение	30
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	31
	ПРИЛОЖЕНИЕ А	33

Введение

В настоящее время большую актуальность имеют системы, предоставляющие информацию о ресурсах операционной системы и частоте системных вызовов. Имея такие сведения, пользователь может проанализировать состояние системы и нагрузку на неё. Особое внимание уделяется операционным системам с ядром Linux [1]. Ядро Linux возможно изучать благодаря тому, что оно имеет открытый исходный код.

На данный момент существует множество различных утилит и команд для получения информации о свободной и занятой оперативной памяти в Linux. Одни из наиболее известных – это команды `free`, `vmstat`, `htop`, `memstat` [2]. Также существует приложение GNOME System Monitor, предоставляющее краткую статистику использования системных ресурсов – памяти, процессора, подкачки и сети – в графическом виде [3].

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием на данную курсовую работу необходимо разработать загружаемый модуль ядра, предоставляющий статистику по количеству доступной и занятой оперативной памяти за выбранный промежуток времени, а также количеству системных вызовов.

Для выполнения поставленной задачи необходимо выполнить следующее:

- проанализировать и сравнить существующие методы и способы решения задачи;
- описать алгоритм решения поставленной задачи и привести соответствующие схемы и IDEF0-диаграммы;
- разработать ПО в соответствии с заданием;
- проанализировать результаты работы разработанного ПО.

Требования к разрабатываемому ПО:

- ПО должно выдавать:
 - количество свободной, доступной и занятой оперативной памяти каждые 10 секунд;
 - количество системных вызовов;
- полученная информация должна записываться в виртуальную файловую систему /proc в файлы /proc/monitor/memory и /proc/monitor/syscalls.

Ограничения на разрабатываемое ПО:

- ПО не предоставляет информацию о том, чем конкретно занята оперативная память;
- промежуток времени, через который фиксируется информация о памяти, задается в коде.

1.2 Информация об оперативной памяти

1.2.1 Файл /proc/meminfo

Одним из способов получения информации об использовании оперативной памяти является файл /proc/meminfo [4]. Из /proc/meminfo можно получить информацию о свободной памяти, об используемой (и физической, и swap), а также о разделяемой (shared memory).

Основные показатели:

- MemTotal – общий объем оперативной памяти;
- LowFree – объем свободной нижней области памяти;
- HighFree – объем свободной верхней области памяти;
- MemFree – сумма LowFree и HighFree;
- MemAvailable – объем доступной оперативной памяти.

Помимо перечисленных в /proc/meminfo присутствует также множество других показателей, не существенных для данной работы.

Основной недостаток использования такого метода решения поставленной задачи заключается в необходимости постоянного обращения напрямую к указанному файлу и его анализа, что может привести к дополнительным временным затратам.

1.2.2 Структура struct sysinfo

Чтобы избежать чтения и анализа файла /proc/meminfo, можно использовать структуру struct sysinfo.

Структура struct sysinfo [5] хранит статистику о всей системе: информацию о времени, прошедшем с начала запуска системы, количество занятой памяти и так далее. В листинге 1 приведено объявление рассматриваемой структуры.

Листинг 1: структура struct sysinfo

```
1 struct sysinfo {  
2     __kernel_long_t uptime;    /* Seconds since boot */  
3     __kernel_ulong_t loads[3]; /* 1, 5, and 15 minute load averages */
```

```

4  __kernel_ulong_t totalram; /* Total usable main memory size */
5  __kernel_ulong_t freeram; /* Free memory size */
6  __kernel_ulong_t sharedram; /* Amount of shared memory */
7  __kernel_ulong_t bufferram; /* Memory used by buffers */
8  __kernel_ulong_t totalswap; /* Total swap space size */
9  __kernel_ulong_t freeswap; /* swap space still available */
10 __u16 procs; /* Number of current processes */
11 __u16 pad; /* Explicit padding for m68k */
12 __kernel_ulong_t totalhigh; /* Total high memory size */
13 __kernel_ulong_t freehigh; /* Available high memory size */
14 __u32 mem_unit; /* Memory unit size in bytes */
15 char _f[20-2*sizeof(__kernel_ulong_t)-sizeof(__u32)]; /* Padding: libc5
    uses this.. */
16 };

```

Наиболее важными полями для данной работы являются: `totalram` (общий объем используемой оперативной памяти) и `freeram` (объем свободной оперативной памяти).

Для инициализации этой структуры используется функция `si_meminfo()`. Стоит отметить, что рассматриваемая структура не содержит информации о доступной памяти в системе. Для того чтобы получить эту информацию, необходимо воспользоваться функцией `si_mem_available()`.

1.2.3 Сравнительный анализ методов

В таблице 1 представлено сравнение рассмотренных методов решения поставленной задачи.

Таблица 1 – Сравнение методов

Метод	Объем свободной памяти	Объем доступной памяти	Объем занятой памяти	Отсутствие необходимости работы с /proc
/proc/meminfo	+	+	+	–
struct sysinfo	+	–	+	+

1.3 Перехват функций

1.3.1 Модификация таблицы системных вызовов

Все обработчики системных вызовов расположены в таблице `sys_call_table`. Подмена значений в этой таблице приведёт к смене поведения всей системы. Сохранив старое значение обработчика и подставив в таблицу собственный обработчик, можно перехватить любой системный вызов.

Особенности данного подхода:

- минимальные временные расходы;
- не требуется специальная конфигурация ядра;
- техническая сложность реализации (поиск таблицы системных вызовов, обход защиты от модификации таблицы, атомарное и безопасное выполнение замены);
- из-за ряда оптимизаций, реализованных в ядре, некоторые обработчики невозможно перехватить [6];
- можно перехватывать только системные вызовы.

1.3.2 kprobes

`kprobes` [7] – 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. Та-

ким образом, kprobes можно использовать как в целях мониторинга, так и для возможности повлиять на дальнейший ход работы ядра.

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре (реализуется с помощью точек останова, внедряемых в исполняемый код ядра);
- для расстановки и обработки точек останова необходимо большое количество процессорного времени [6];
- техническая сложность реализации (чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте стека они находятся).

1.3.3 Kernel tracepoints

Kernel tracepoints [8] – это фреймворк для трассировки ядра, реализованный через статическое инструментирование кода (т. е. выполняемое однократно перед запуском программы).

Особенности рассматриваемого фреймворка:

- минимальные накладные расходы – необходимо только вызвать функцию трассировки в соответствующем месте;
- не все функции ядра статически инструментированы;
- не работает, если ядро не сконфигурировано должным образом [6].

1.3.4 ftrace

ftrace [9] – это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора -pg и -mfentry, которые вставляют в начало каждой функции вызов специальной трассировочной функции mcount() или __fentry__().

Для большинства современных архитектур процессора доступна оптимизация: динамический ftrace [10]. Ядро знает расположение всех вызовов функций mcount() или __fentry__() и на ранних этапах загрузки ядра подменяет их машинный код на инструкцию NOP. При включении трассировки вызовы ftrace добавляются обратно в соответствующие функции.

Особенности рассматриваемого фреймворка:

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

1.3.5 Сравнительный анализ методов

В таблице 2 приведено сравнение рассмотренных методов.

Таблица 2 – Методы перехвата системных вызовов

Название	Дин. за- грузка	Перехват любых функций	Любая конфи- гурация ядра	Про- стота реализа- ции
Модификация таблицы систем- ных вызовов	+	—	+	—
kprobes	+	+	+	—
kernel tracepoints	+	+	—	+
ftrace	+	+	—	+

Выводы

В этом разделе была проанализирована поставленная задача и методы ее решения. В ходе анализа для получения информации о памяти была выбрана структура `struct sysinfo`, т. к. такой подход позволяет не работать напрямую с файлом `/proc/meminfo`. При этом для получения информации о доступной памяти будет использоваться функция `si_mem_available()`.

Для перехвата функций был выбран фреймворк `ftrace`, так как он позволяет перехватить любую функцию, зная лишь её имя, может быть загружен в ядро динамически и не требует специальной сборки ядра.

2 Конструкторская часть

2.1 Структура `ftrace_ops` и функция коллбека

Для регистрации функции коллбека необходима структура `ftrace_ops` [11].

Структура приведена в листинге 2.

Листинг 2: структура `ftrace_ops`

```
1  struct ftrace_ops ops = {
2      .func      = callback_func,
3      .flags      = FTRACE_FLAGS
4      .private    = any_private_data_structure,
5  };
```

Эта структура используется, чтобы сообщить `ftrace`, какую функцию следует вызывать в качестве коллбека, а также какие меры защиты будут выполняться коллбеком. Поля `flags` и `private` являются необязательными.

Включение отслеживания вызовов:

```
1  register_ftrace_function(&ops);
```

Отключение отслеживания вызовов:

```
1  unregister_ftrace_function(&ops);
```

Прототип функции коллбека выглядит следующим образом:

```
1  void callback_func(unsigned long ip, unsigned long parent_ip, struct
    ftrace_ops *op, struct pt_regs *regs);
```

- `ip` – указатель инструкции перехватываемой функции;
- `parent_ip` – указатель инструкции функции, вызвавшей перехватываемую функцию;
- `op` – указатель на `ftrace_ops`;
- `regs` – если в структуре `ftrace_ops` установлены флаги

`FTRACE_OPS_FL_SAVE_REGS` или

`FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED`, то это будет указывать

на структуру `pt_regs`, как если бы точка останова была размещена в начале функции, которую перехватывал `ftrace`. В противном случае он либо содержит мусор, либо `NULL`.

2.2 Алгоритм перехвата системного вызова

На рисунке 1 представлена схема алгоритма перехвата системных вызовов на примере `sys_clone`.

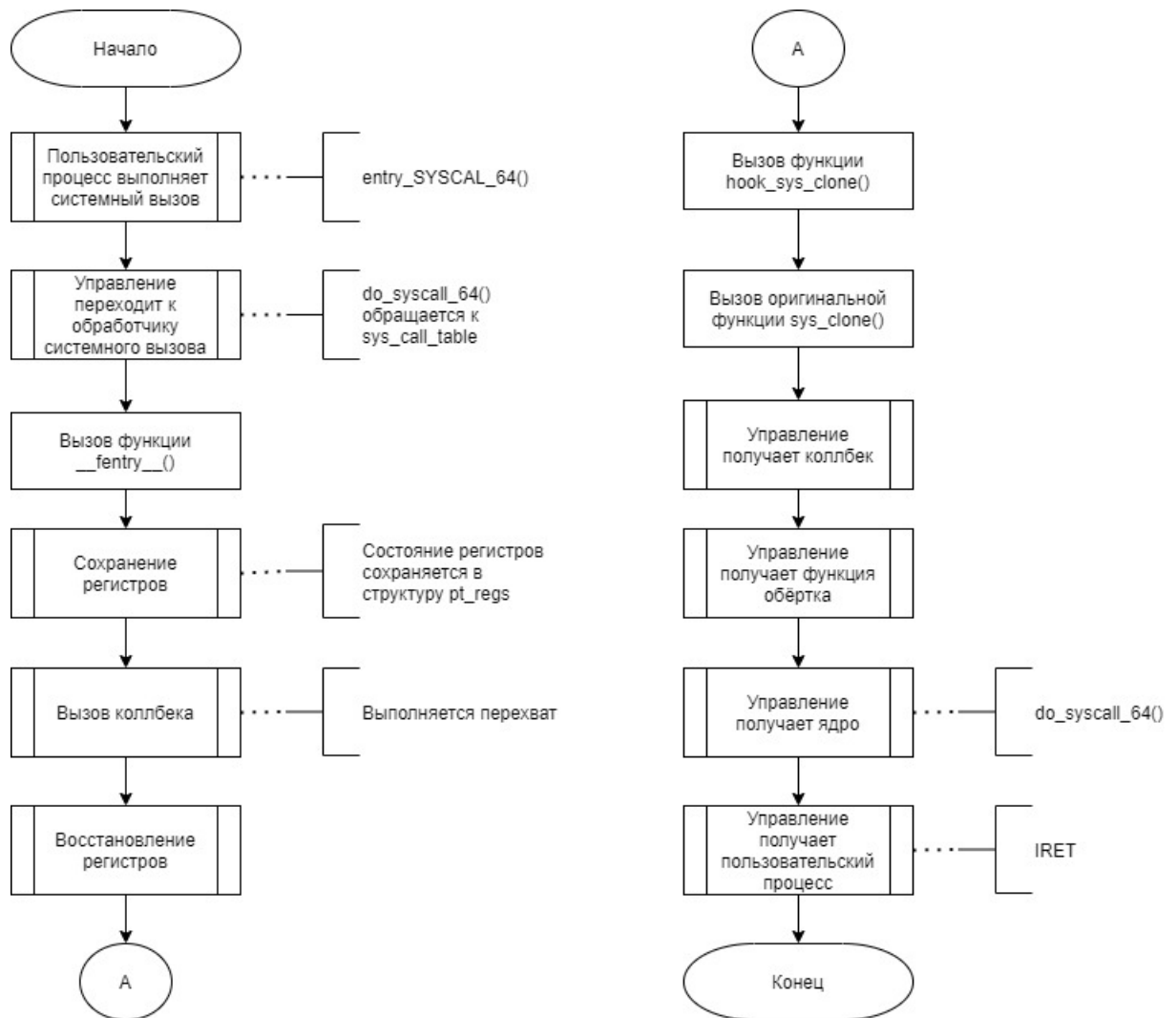


Рисунок 1 – Алгоритм перехвата системного вызова

- 1) Пользовательский процесс выполняет инструкцию SYSCALL. С помощью этой инструкции выполняется переход в режим ядра и управление передаётся низкоуровневому обработчику системных вызовов `entry_SYSCALL_64()`. Этот обработчик отвечает за все системные вызовы 64-битных программ на 64-битных машинах.
- 2) 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 – `sys_clone()`.
- 3) Вызывается `ftrace`. В начале каждой функции ядра находится вызов функции `__fentry__()`, реализованная фреймворком `ftrace`. Перед этим состояние регистров сохраняется в специальную структуру `pt_regs`.
- 4) `ftrace` вызывает разработанный коллбек.
- 5) Коллбек выполняет перехват. Коллбек анализирует значение `parent_ip` и выполняет перехват, обновляя значение регистра `rip` (указатель на следующую исполняемую инструкцию) в структуре `pt_regs`.
- 6) `ftrace` восстанавливает значение регистров с помощью структуры `pt_regs`. Так как обработчик изменяет значение регистра `rip` – это приведёт к передаче управления по новому адресу.
- 7) Управление получает функция обёртка. 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()`.
- 8) Функция обёртка вызывает оригинальную функцию. Функция `hook_sys_clone()` может проанализировать аргументы и контекст системного вызова и запретить или разрешить процессу его выполнение. В случае его запрета, функция просто возвращает код ошибки. Иначе – вызывает

ет оригинальный обработчик `sys_clone()` повторно, с помощью указателя `real_sys_clone`, который был сохранён при настройке перехвата.

- 9) Управление получает коллбек. Как и при первом вызове `sys_clone()`, управление проходит через `ftrace` и передается в коллбек.
- 10) Коллбек ничего не делает. В этот раз функция `sys_clone()` вызывается разработанной функцией `hook_sys_clone()`, а не ядром из функции `do_syscall_64()`. Коллбек не модифицирует регистры и выполнение функции `sys_clone()` продолжается как обычно.
- 11) Управление передаётся функции обёртке.
- 12) Управление передаётся ядру. Функция `hook_sys_clone()` завершается и управление переходит к `do_syscall_64()`.
- 13) Управление возвращает в пользовательский процесс. Ядро выполняет инструкцию `IRET`, устанавливая регистры для нового пользовательского процесса и переводя центральный процессор в режим исполнения пользовательского кода.

2.3 Алгоритм подсчёта количества системных вызовов

На рисунке 2 представлена схема алгоритма подсчёта системных вызовов.



Рисунок 2 – Алгоритм подсчёта количества системных вызовов

- Агрегирующий массив – это массив на 86400 элементов (что соответствует 24 часам), состоящий из структур, имеющих два поля в виде 64-битных беззнаковых целых чисел. Это позволяет фиксировать до 128 системных вызовов в секунду на протяжении 24 часов. Такой массив занимает всего лишь 1350 килобайт оперативной памяти;
- спин-блокировка необходима с той целью, что несколько системных вызовов могут быть вызваны в один и тот же момент времени – в таком случае, без блокировки, агрегирующий массив потеряет часть данных.

2.4 Алгоритм получения информации об объеме памяти

На рисунке 3 представлена схема алгоритма работы потока ядра, предназначенного для подсчета объема свободной и занятой оперативной памяти в системе.

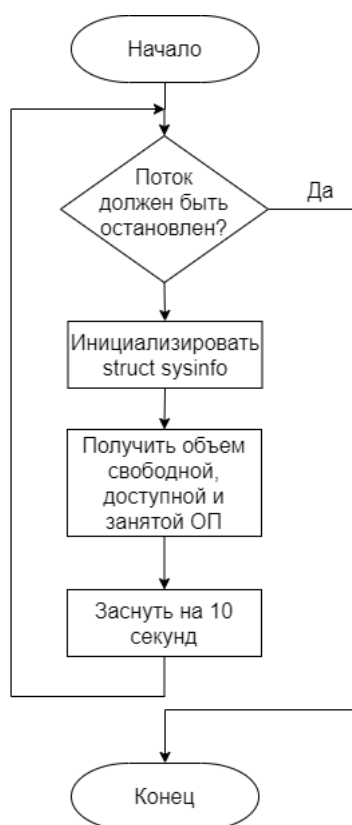


Рисунок 3 – Схема алгоритма работы потока

2.5 Точки входа

Точками входа разрабатываемого загружаемого модуля ядра будут являться функции инициализации и выхода из модуля.

При инициализации модуля будут создаваться необходимые для сохранения результатов директория и файлы в /proc, происходить создание и запуск потока ядра для подсчета объема оперативной памяти и устанавливаться хуки для перехвата системных вызовов.

Функция выхода из модуля в свою очередь будет предназначена для остановки потока, удаления созданных директории и файлов в /proc и удаления установленных хуков.

3 Технологическая часть

3.1 Выбор языка программирования

В качестве языка программирования был выбран язык C [12]. Данный выбор обусловлен тем, что исходный код ядра Linux, все его модули и драйверы написаны на этом языке.

В качестве компилятора был выбран gcc [13].

3.2 Информация о памяти в системе

Для сбора информации о доступной и свободной памяти в системе запускается отдельный поток ядра, который находится в состоянии сна и, просыпаясь каждые 10 секунд, фиксирует эту информацию в результирующий массив. В листинге 3 представлена реализация этого потока, а в листинге 4 его инициализация.

Листинг 3: реализация функции сохраняющей информацию о памяти

```
1  mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
2  int mem_info_calls_cnt;
3
4  int memory_cnt_task_handler_fn(void *args) {
5      struct sysinfo i;
6      struct timespec64 t;
7
8      ENTER_LOG();
9
10     allow_signal(SIGKILL);
11
12     while (!kthread_should_stop()) {
13         si_meminfo(&i);
14
15         ktime_get_real_ts64(&t);
16
17         mem_info_array[mem_info_calls_cnt].free = i.freeram;
18         mem_info_array[mem_info_calls_cnt].available = si_mem_available();
19         mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
20
21         ssleep(10);
```

```

22
23     if (signal_pending(worker_task)) {
24         break;
25     }
26 }
27
28 EXIT_LOG();
29 do_exit(0);
30 return 0;
31 }

```

Листинг 4: инициализация потока ядра

```

1  cpu = get_cpu();
2  worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
   counter thread");
3  kthread_bind(worker_task, cpu);
4
5  if (worker_task == NULL) {
6      cleanup();
7      return -1;
8  }
9
10 wake_up_process(worker_task);
11 return 0;

```

3.3 Поиск адреса перехватываемой функции

Для корректной работы ftrace необходимо найти и сохранить адрес функции, которую будет перехватывать разрабатываемый модуль ядра.

В старых версиях ядра (в версии ядра 5.7.0 данная функция перестала быть экспортируемой [14]) найти адрес функции можно было с помощью функции `kallsyms_lookup_name()` – списка всех символов в ядре, в том числе не экспортируемых для модулей. Так как модуль ядра разрабатывался на системе с версией ядра 5.15.0, воспользоваться данным способом было нельзя. Поэтому использовался интерфейс `kprobes`.

Из-за того, что данный способ имеет больше накладных расходов, чем

поиск с помощью `kallsyms_lookup_name()` (требуется регистрация и удаление `kprobes` в системе), для версий ядра ниже 5.7.0 поиск адреса производится с помощью `kallsyms_lookup_name()`.

Реализация функции `lookup_name()`, возвращающей адрес перехватываемой функции по её названию, представлена в листинге 5.

Листинг 5: 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      ENTER_LOG();
10
11     if (register_kprobe(&kp) < 0) {
12         EXIT_LOG();
13         return 0;
14     }
15
16     retval = (unsigned long) kp.addr;
17     unregister_kprobe(&kp);
18
19     EXIT_LOG();
20
21     return retval;
22 }
23 #else
24 static unsigned long lookup_name(const char *name)
25 {
26     unsigned long retval;
27
28     ENTER_LOG();
29     retval = kallsyms_lookup_name(name);
30     EXIT_LOG();
31
32     return retval;
```

```
33  }
34  #endif
```

3.4 Инициализация ftrace

В листинге 6 представлена реализация функции, которая инициализирует структуру `ftrace_ops`.

Листинг 6: Реализация функции `install_hook()`

```
1  static int install_hook(struct ftrace_hook *hook) {
2      int rc;
3
4      ENTER_LOG();
5
6      if ((rc = resolve_hook_address(hook))) {
7          EXIT_LOG();
8          return rc;
9      }
10
11     hook->ops.func = ftrace_thunk;
12     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
13         | FTRACE_OPS_FL_RECURSION
14         | FTRACE_OPS_FL_IPMODIFY;
15
16     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0))) {
17         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
18         return rc;
19     }
20
21     if ((rc = register_ftrace_function(&hook->ops))) {
22         pr_debug("register_ftrace_function() failed: %d\n", rc);
23         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
24     }
25
26     EXIT_LOG();
27
28     return rc;
29 }
```

В листинге 7 представлена реализация отключения перехвата функции.

Листинг 7: Реализация функции `remove_hook()`

```
1  static void remove_hook(struct ftrace_hook *hook) {
2      int rc;
3
4      ENTER_LOG();
5
6      if (hook->address == 0x00) {
7          EXIT_LOG();
8          return;
9      }
10
11     if ((rc = unregister_ftrace_function(&hook->ops))) {
12         pr_debug("unregister_ftrace_function() failed: %d\n", rc);
13     }
14
15     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0))) {
16         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
17     }
18
19     hook->address = 0x00;
20
21     EXIT_LOG();
22 }
```

3.5 Функции обёртки

При объявлении функций обёрток, которые будут запущены вместо перехватываемой функции, необходимо в точности соблюдать сигнатуру. Оригинальные описания функций были взяты из исходных кодов ядра Linux.

В листинге 8 представлена реализация функции обёртки на примере `sys_clone()`.

Листинг 8: Реализация функции обёртки

```
1  static asmlinkage long (*real_sys_clone)(unsigned long clone_flags,
2      unsigned long newsp, int __user *parent_tidptr,
3      int __user *child_tidptr, unsigned long tls);
4
5  static asmlinkage long hook_sys_clone(unsigned long clone_flags,
6      unsigned long newsp, int __user *parent_tidptr,
7      int __user *child_tidptr, unsigned long tls)
```

```

8  {
9      update_syscall_array(SYS_CLONE_NUM);
10     return real_sys_clone(clone_flags, newsp, parent_tidptr, child_tidptr,
        tls);
11 }

```

В листинге 9 представлена реализация функции которая обновляет массив, хранящий количество системных вызовов за последние 24 часа.

Листинг 9: Реализация функции `update_syscall_array()`

```

1  static DEFINE_SPINLOCK(my_lock);
2
3  static void inline update_syscall_array(int syscall_num) {
4      ktime_t time;
5
6      time = ktime_get_boottime_seconds() - start_time;
7
8      spin_lock(&my_lock);
9
10     if (syscall_num < 64) {
11         syscalls_time_array[time % TIME_ARRAY_SIZE].p1 |= 1UL << syscall_num;
12     } else {
13         syscalls_time_array[time % TIME_ARRAY_SIZE].p2 |= 1UL << (syscall_num %
            64);
14     }
15
16     spin_unlock(&my_lock);
17 }

```

3.6 Получение информации о количестве системных вызовов

В листинге 10 представлена реализация функций, которые агрегируют информацию о системных вызовах (данные массива `update_syscall_array`) и предоставляют ее в читаемом для пользователя виде.

Листинг 10: Реализация функций агрегации данных о системных вызовах

```

1  static inline void walk_bits_and_find_syscalls(struct seq_file *m, uint64_t
        num, int syscalls_arr_cnt[]) {
2      int i;

```

```

3
4     for (i = 0; i < 64; i++) {
5         if (num & (1UL << i)) {
6             syscalls_arr_cnt[i]++;
7         }
8     }
9 }
10
11 void print_syscall_statistics(struct seq_file *m, const ktime_t mstart,
12                             ktime_t range) {
13     int syscalls_arr_cnt[128];
14     uint64_t tmp;
15     size_t i;
16     ktime_t uptime;
17
18     memset((void*)syscalls_arr_cnt, 0, 128 * sizeof(int));
19     uptime = ktime_get_boottime_seconds() - mstart;
20
21     if (uptime < range) {
22         range = uptime;
23     }
24
25     for (i = 0; i < range; i++) {
26         if ((tmp = syscalls_time_array[uptime - i].p1) != 0) {
27             walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt);
28         }
29
30         if ((tmp = syscalls_time_array[uptime - i].p2) != 0) {
31             walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt + 64);
32         }
33     }
34
35     show_int_message(m, "Syscall statistics for the last %d seconds.\n\n",
36                     range);
37
38     for (i = 0; i < 128; i++) {
39         if (syscalls_arr_cnt[i] != 0) {
40             show_str_message(m, "%s called ", syscalls_names[i]);
41             show_int_message(m, "%d times.\n", syscalls_arr_cnt[i]);
42         }
43     }
44 }

```

```
41     }  
42 }
```

3.7 Детали реализации

В листингах 11-13 представлена реализация точек входа в загружаемый модуль.

Листинг 11: функция инициализации модуля

```
1  static int __init md_init(void) {  
2      int rc;  
3      int cpu;  
4  
5      ENTER_LOG();  
6  
7      if ((rc = proc_init())) {  
8          return rc;  
9      }  
10  
11     if ((rc = install_hooks())) {  
12         cleanup();  
13         return rc;  
14     }  
15  
16     start_time = ktime_get_boottime_seconds();  
17  
18     cpu = get_cpu();  
19     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory  
counter thread");  
20     kthread_bind(worker_task, cpu);  
21  
22     if (worker_task == NULL) {  
23         cleanup();  
24         return -1;  
25     }  
26  
27     wake_up_process(worker_task);  
28  
29     printk("%s: module loaded\n", MODULE_NAME);  
30     EXIT_LOG();  
}
```



```
31
32     return 0;
33 }
```

Листинг 12: функция выхода из модуля

```
1  static void __exit md_exit(void) {
2      cleanup();
3
4      printk("%s: module unloaded\n", MODULE_NAME);
5  }
```

Листинг 13: функция cleanup()

```
1  static void cleanup(void) {
2      ENTER_LOG();
3
4      if (worker_task) {
5          kthread_stop(worker_task);
6      }
7
8      if (proc_mem_file != NULL) {
9          remove_proc_entry("memory", proc_root);
10     }
11
12     if (proc_syscall_file != NULL) {
13         remove_proc_entry("syscalls", proc_root);
14     }
15
16     if (proc_root != NULL) {
17         remove_proc_entry(MODULE_NAME, NULL);
18     }
19
20     remove_hooks();
21
22     EXIT_LOG();
23 }
```

Make файл для компиляции и сборки разработанного загружаемого модуля ядра представлен в листинге 14.

Листинг 14: реализация make файла

```
1  KPATH := /lib/modules/$(shell uname -r)/build
2  MDIR  := $(shell pwd)
3
4  obj-m += monitor.o
5  monitor-y := monitor_main.o stat.o hooks.o log.o
6  EXTRA_CFLAGS=-I$(PWD)/inc
7
8  all:
9  make -C $(KPATH) M=$(MDIR) modules
10
11  clean:
12  make -C $(KPATH) M=$(MDIR) clean
13
14  load:
15  sudo insmod monitor.ko
16
17  unload:
18  sudo rmmod monitor.ko
19
20  info:
21  modinfo monitor.ko
22
23  logs:
24  sudo dmesg | tail -n60 | grep monitor:
```

4 Исследовательская часть

4.1 Результаты работы разработанного ПО

На рисунках 4-5 представлены примеры работы модуля, а на рисунке 6 – сравнение результатов получения информации об объеме оперативной памяти с результатами команды «free -m» (у последней все объемы указаны в Мб).

```
Time 00:22:11
Free:                3687544 kB
Available:           5390508 kB
Occupied:            2543792 kB

Time 00:22:22
Free:                3676760 kB
Available:           5379732 kB
Occupied:            2554568 kB

Time 00:22:32
Free:                4506996 kB
Available:           6207136 kB
Occupied:            1727164 kB

Time 00:22:42
Free:                4503000 kB
Available:           6203140 kB
Occupied:            1731160 kB

Time 00:22:52
Free:                4510588 kB
Available:           6210736 kB
Occupied:            1723564 kB
a_avortep@avortep:~/prog/os_course/src$
```

Рисунок 4 – Информация об оперативной памяти в системе

```
a_avortep@avortep:~/prog/os_course/src$ cat /proc/monitor/syscalls
Syscall statistics for the last 192 seconds.

sys_read called 192 times.
sys_write called 192 times.
sys_close called 135 times.
sys_mmap called 96 times.
sys_sched_yield called 4 times.
sys_socket called 19 times.
sys_connect called 14 times.
sys_accept called 4 times.
sys_sendto called 27 times.
sys_recvfrom called 32 times.
sys_sendmsg called 95 times.
sys_recvmsg called 177 times.
sys_shutdown called 7 times.
sys_clone called 21 times.
sys_execve called 8 times.
sys_mkdir called 12 times.
sys_rmdir called 6 times.
```

Рисунок 5 – Информация о количестве системных вызовов за последние 192 секунды

```

Time 00:31:35
Free: 3896304 kB
Available: 6211480 kB
Occupied: 1722820 kB

Time 00:31:45
Free: 3886056 kB
Available: 6201256 kB
Occupied: 1733044 kB

Time 00:31:55
Free: 4016820 kB
Available: 6332388 kB
Occupied: 1601912 kB
a_avortep@avortep:~/prog/os_course/src$ free -m

```

	total	used	free	shared	buff/cache	available
Mem:	7748	1062	3922	220	2763	6184
Swap:	7812	0	7812			

```

a_avortep@avortep:~/prog/os_course/src$

```

Рисунок 6 – Результаты работы команды «free -m»

На рисунке 7 представлена визуализация данных о свободной, доступной и занятой памяти в системе, полученных из разработанного модуля ядра.



Рисунок 7 – Визуализация данных о памяти за 20 минут

4.2 Анализ результатов

На основе рисунка 6 можно сделать вывод, что результаты работы разработанного загружаемого модуля ядра совпадают с результатами работы команды «free -m». Следовательно, результаты корректны.

По рисунку 7 видно, как изменялась загруженность памяти в течение 20 минут. В моменты резкого спада количества занятой оперативной памяти было закрыто несколько приложений, а в моменты роста – наоборот, открыто.

Заключение

Цель курсового проекта достигнута. В ходе проделанной работы был разработан загружаемый модуль ядра, предоставляющий информацию о загрузенности системы: количество системных вызовов за выбранный промежуток времени, а также количество свободной и доступной оперативной памяти.

Для этого были изучены структуры и функции ядра, которые предоставляют информацию о памяти, и проанализированы существующие подходы к перехвату системных вызовов. На основе чего был разработан, а затем реализован алгоритм работы программы.

Помимо этого были также приведены и проанализированы результаты работы ПО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux – Operating System [Электронный ресурс]. – Режим доступа: <https://www.linux.org/> (дата обращения: 20.12.2022).
2. Использование оперативной памяти в Linux [Электронный ресурс]. – Режим доступа: <https://losst.pro/ispolzovanie-operativnoj-pamyati-linux> (дата обращения: 20.12.2022).
3. System Monitor – Apps for GNOME [Электронный ресурс]. – Режим доступа: <https://apps.gnome.org/ru/app/gnome-system-monitor/> (дата обращения: 23.12.2022).
4. The /proc/meminfo File in Linux [Электронный ресурс]. – Режим доступа: <https://www.baeldung.com/linux/proc-meminfo> (дата обращения: 24.12.2022).
5. include/uapi/linux/sysinfo.h - Linux source code (v5.15) [Электронный ресурс]. – Режим доступа: <https://elixir.bootlin.com/linux/v5.15/source/include/uapi/linux/sysinfo.h> (дата обращения: 24.12.2022).
6. Механизмы профилирования Linux [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/company/metrotek/blog/261003/> (дата обращения: 13.02.2023).
7. Kernel Probes (Kprobes) [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/html/latest/trace/kprobes.html> (дата обращения: 14.02.2023).
8. Using the Linux Kernel Tracepoints [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/html/latest/trace/tracepoints.html> (дата обращения: 14.02.2023).

9. Using ftrace | Android Open Source Project [Электронный ресурс]. – Режим доступа: <https://source.android.com/devices/tech/debug/ftrace> (дата обращения: 14.02.2023).
10. Трассировка ядра с ftrace [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/company/selectel/blog/280322/> (дата обращения: 14.02.2023).
11. Using ftrace to hook to functions [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/html/latest/trace/ftrace-uses.html> (дата обращения: 15.02.2023).
12. C99 standard note [Электронный ресурс]. – Режим доступа: <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> (дата обращения: 17.01.2023).
13. GCC, the GNU Compiler Collection [Электронный ресурс]. – Режим доступа: <https://gcc.gnu.org/> (дата обращения: 17.01.2023).
14. Unexporting kallsyms_lookup_name() [Электронный ресурс]. – Режим доступа: <https://lwn.net/Articles/813350/> (дата обращения: 15.02.2023).

ПРИЛОЖЕНИЕ А

Листинг 15: листинг файла monitor main.c

```
1  #include <linux/module.h>
2  #include <linux/proc_fs.h>
3  #include <linux/time.h>
4  #include <linux/kthread.h>
5
6  #include "hooks.h"
7  #include "memory.h"
8  #include "stat.h"
9
10 MODULE_LICENSE("GPL");
11 MODULE_AUTHOR("Petrova Anna");
12 MODULE_DESCRIPTION("A utility for monitoring RAM usage and number of
    syscalls");
13
14 static struct proc_dir_entry *proc_root = NULL;
15 static struct proc_dir_entry *proc_mem_file = NULL, *proc_syscall_file =
    NULL;
16 static struct task_struct *worker_task = NULL;
17
18 extern ktime_t start_time;
19 /* default syscall range value is 10 min */
20 static ktime_t syscalls_range_in_seconds = 600;
21
22 static int show_memory(struct seq_file *m, void *v) {
23     print_memory_statistics(m);
24     return 0;
25 }
26
27 static int proc_memory_open(struct inode *sp_inode, struct file *sp_file) {
28     return single_open(sp_file, show_memory, NULL);
29 }
30
31 static int show_syscalls(struct seq_file *m, void *v) {
32     print_syscall_statistics(m, start_time, syscalls_range_in_seconds);
33     return 0;
34 }
```

```

35
36 static int proc_syscalls_open(struct inode *sp_inode, struct file *sp_file)
37 {
38     return single_open(sp_file, show_syscalls, NULL);
39 }
40
41 static int proc_release(struct inode *sp_node, struct file *sp_file) {
42     return 0;
43 }
44
45 mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
46
47 int mem_info_calls_cnt;
48
49 int memory_cnt_task_handler_fn(void *args) {
50     struct sysinfo i;
51     struct timespec64 t;
52
53     ENTER_LOG();
54
55     allow_signal(SIGKILL);
56
57     while (!kthread_should_stop()) {
58         si_meminfo(&i);
59
60         ktime_get_real_ts64(&t);
61
62         mem_info_array[mem_info_calls_cnt].free = i.freeram;
63         mem_info_array[mem_info_calls_cnt].available = si_mem_available();
64         mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
65
66         ssleep(10);
67
68         if (signal_pending(worker_task)) {
69             break;
70         }
71     }
72
73     EXIT_LOG();
74     do_exit(0);
75     return 0;

```

```

74  }
75
76  #define CHAR_TO_INT(ch) (ch - '0')
77
78  static ktime_t convert_strf_to_seconds(char buf[]) {
79      /* time format: xxhyyzzs. For example: 01h23m45s */
80      ktime_t hours, min, secs;
81
82      hours = CHAR_TO_INT(buf[0]) * 10 + CHAR_TO_INT(buf[1]);
83      min = CHAR_TO_INT(buf[3]) * 10 + CHAR_TO_INT(buf[4]);
84      secs = CHAR_TO_INT(buf[6]) * 10 + CHAR_TO_INT(buf[7]);
85
86      return hours * 60 * 60 + min * 60 + secs;
87  }
88
89  static ssize_t proc_syscall_write(struct file *file, const char __user *buf
    , size_t len, loff_t *ppos) {
90      char syscalls_time_range[10];
91
92      ENTER_LOG();
93
94      if (copy_from_user(&syscalls_time_range, buf, len) != 0) {
95          EXIT_LOG();
96          return -EFAULT;
97      }
98
99      syscalls_range_in_seconds = convert_strf_to_seconds(syscalls_time_range);
100
101      EXIT_LOG();
102      return len;
103  }
104
105  static const struct proc_ops mem_ops = {
106      proc_read: seq_read,
107      proc_open: proc_memory_open,
108      proc_release: proc_release,
109  };
110
111  static const struct proc_ops syscalls_ops = {
112      proc_read: seq_read,

```

```

113     proc_open: proc_syscalls_open,
114     proc_release: proc_release,
115     proc_write: proc_syscall_write,
116 };
117
118 static void cleanup(void) {
119     ENTER_LOG();
120
121     if (worker_task) {
122         kthread_stop(worker_task);
123     }
124
125     if (proc_mem_file != NULL) {
126         remove_proc_entry("memory", proc_root);
127     }
128
129     if (proc_syscall_file != NULL) {
130         remove_proc_entry("syscalls", proc_root);
131     }
132
133     if (proc_root != NULL) {
134         remove_proc_entry(MODULE_NAME, NULL);
135     }
136
137     remove_hooks();
138
139     EXIT_LOG();
140 }
141
142 static int proc_init(void) {
143     ENTER_LOG();
144
145     if ((proc_root = proc_mkdir(MODULE_NAME, NULL)) == NULL) {
146         goto err;
147     }
148
149     if ((proc_mem_file = proc_create("memory", 066, proc_root, &mem_ops)) ==
        NULL) {
150         goto err;
151     }

```

```

152
153     if ((proc_syscall_file = proc_create("syscalls", 066, proc_root, &
154         syscalls_ops)) == NULL)
155     {
156         goto err;
157     }
158     EXIT_LOG();
159     return 0;
160
161     err:
162     cleanup();
163     EXIT_LOG();
164     return -ENOMEM;
165 }
166
167 static int __init md_init(void) {
168     int rc;
169     int cpu;
170
171     ENTER_LOG();
172
173     if ((rc = proc_init())) {
174         return rc;
175     }
176
177     if ((rc = install_hooks())) {
178         cleanup();
179         return rc;
180     }
181
182     start_time = ktime_get_boottime_seconds();
183
184     cpu = get_cpu();
185     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
186         counter thread");
187     kthread_bind(worker_task, cpu);
188
189     if (worker_task == NULL) {
190         cleanup();

```

```

190     return -1;
191 }
192
193 wake_up_process(worker_task);
194
195 printk("%s: module loaded\n", MODULE_NAME);
196 EXIT_LOG();
197
198 return 0;
199 }
200
201 static void __exit md_exit(void) {
202     cleanup();
203
204     printk("%s: module unloaded\n", MODULE_NAME);
205 }
206
207 module_init(md_init);
208 module_exit(md_exit);

```

Листинг 16: листинг файла stat.c

```

1  #include "stat.h"
2
3  static inline long convert_to_kb(const long n) {
4      return n << (PAGE_SHIFT - 10);
5  }
6
7  void print_memory_statistics(struct seq_file *m) {
8      struct sysinfo info;
9      long long secs;
10     long sys_occupied;
11     int i;
12
13     ENTER_LOG();
14
15     si_meminfo(&info);
16     show_int_message(m, "Memory total: \t%ld kB\n", convert_to_kb(info.
totalram));
17
18     for (i = 0; i < mem_info_calls_cnt; i++) {

```

```

19     secs = mem_info_array[i].time_secs;
20     show_int3_message(m, "\nTime %.2llu:%.2llu:%.2llu\n", (secs / 3600 + 3)
    % 24, secs / 60 % 60, secs % 60);
21     show_int_message(m, "Free:      \t\t\t%ld kB\n", convert_to_kb(
    mem_info_array[i].free));
22     show_int_message(m, "Available: \t\t\t%ld kB\n", convert_to_kb(
    mem_info_array[i].available));
23     sys_occupied = convert_to_kb(info.totalram) - convert_to_kb(
    mem_info_array[i].available);
24     show_int_message(m, "Occupied: \t%ld kB\n", sys_occupied);
25 }
26
27 EXIT_LOG();
28 }
29
30 syscalls_info_t syscalls_time_array[TIME_ARRAY_SIZE];
31
32 static const char *syscalls_names[] = {
33     "sys_read", // 0
34     "sys_write",
35     "sys_open",
36     "sys_close",
37     "sys_newstat", // 4
38     "sys_newfstat",
39     "sys_newlstat",
40     "sys_poll",
41     "sys_lseek",
42     "sys_mmap", // 9
43     "sys_mprotect",
44     "sys_munmap",
45     "sys_brk",
46     "sys_rt_sigaction",
47     "sys_rt_sigprocmask", // 14
48     "stub_rt_sigreturn",
49     "sys_ioctl",
50     "sys_pread64",
51     "sys_pwrite64",
52     "sys_readv", // 19
53     "sys_writev",
54     "sys_access",

```

```
55     "sys_pipe",
56     "sys_select",
57     "sys_sched_yield", // 24
58     "sys_mremap",
59     "sys_msync",
60     "sys_mincore",
61     "sys_madvise",
62     "sys_shmget", // 29
63     "sys_shmat",
64     "sys_shmctl",
65     "sys_dup",
66     "sys_dup2",
67     "sys_pause", // 34
68     "sys_nanosleep",
69     "sys_gettimer",
70     "sys_alarm",
71     "sys_settimer",
72     "sys_getpid", // 39
73     "sys_sendfile64",
74     "sys_socket",
75     "sys_connect",
76     "sys_accept",
77     "sys_sendto", // 44
78     "sys_recvfrom",
79     "sys_sendmsg",
80     "sys_recvmsg",
81     "sys_shutdown",
82     "sys_bind", // 49
83     "sys_listen",
84     "sys_getsockname",
85     "sys_getpeername",
86     "sys_socketpair",
87     "sys_setsockopt", // 54
88     "sys_getsockopt",
89     "sys_clone",
90     "sys_fork",
91     "sys_vfork",
92     "sys_execve", // 59
93     "sys_exit",
94     "sys_wait4",
```



```
95     "sys_kill",
96     "sys_newuname",
97     "sys_semget", // 64
98     "sys_semop",
99     "sys_semctl",
100    "sys_shmdt",
101    "sys_msgget",
102    "sys_msgsnd", // 69
103    "sys_msgrcv",
104    "sys_msgctl",
105    "sys_fcntl",
106    "sys_flock",
107    "sys_fsync", // 74
108    "sys_fdatasync",
109    "sys_truncate",
110    "sys_ftruncate",
111    "sys_getdents",
112    "sys_getcwd", // 79
113    "sys_chdir",
114    "sys_fchdir",
115    "sys_rename",
116    "sys_mkdir",
117    "sys_rmdir", // 84
118    "sys_creat",
119    "sys_link",
120    "sys_unlink",
121    "sys_symlink",
122    "sys_readlink", // 89
123    "sys_chmod",
124    "sys_fchmod",
125    "sys_chown",
126    "sys_fchown",
127    "sys_lchown", // 94
128    "sys_umask",
129    "sys_gettimeofday",
130    "sys_getrlimit",
131    "sys_getrusage",
132    "sys_sysinfo", // 99
133    "sys_times",
134    "sys_ptrace",
```

```

135     "sys_getuid",
136     "sys_syslog",
137     "sys_getgid", // 104
138     "sys_setuid",
139     "sys_setgid",
140     "sys_geteuid",
141     "sys_getegid",
142     "sys_getpgid", // 109
143     "sys_getppid",
144     "sys_getpgrp",
145     "sys_setsid",
146     "sys_setreuid",
147     "sys_setregid", // 114
148     "sys_getgroups",
149     "sys_setgroups",
150     "sys_setresuid",
151     "sys_getresuid",
152     "sys_setresgid", // 119
153     "sys_getresgid",
154     "sys_getpgrp",
155     "sys_setfsuid",
156     "sys_setfsgid",
157     "sys_getsid", // 124
158     "sys_capget",
159     "sys_capset",
160     "sys_rt_sigpending", // 127
161 };
162
163 static inline void walk_bits_and_find_syscalls(struct seq_file *m, uint64_t
        num, int syscalls_arr_cnt[]) {
164     int i;
165
166     for (i = 0; i < 64; i++) {
167         if (num & (1UL << i)) {
168             syscalls_arr_cnt[i]++;
169         }
170     }
171 }
172
173 void print_syscall_statistics(struct seq_file *m, const ktime_t mstart,

```

```

        ktime_t range) {
174     int syscalls_arr_cnt[128];
175     uint64_t tmp;
176     size_t i;
177     ktime_t uptime;
178
179     memset((void*)syscalls_arr_cnt, 0, 128 * sizeof(int));
180     uptime = ktime_get_boottime_seconds() - mstart;
181
182     if (uptime < range) {
183         range = uptime;
184     }
185
186     for (i = 0; i < range; i++) {
187         if ((tmp = syscalls_time_array[uptime - i].p1) != 0) {
188             walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt);
189         }
190
191         if ((tmp = syscalls_time_array[uptime - i].p2) != 0) {
192             walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt + 64);
193         }
194     }
195
196     show_int_message(m, "Syscall statistics for the last %d seconds.\n\n",
range);
197
198     for (i = 0; i < 128; i++) {
199         if (syscalls_arr_cnt[i] != 0) {
200             show_str_message(m, "%s called ", syscalls_names[i]);
201             show_int_message(m, "%d times.\n", syscalls_arr_cnt[i]);
202         }
203     }
204 }

```

Листинг 17: листинг файла log.c

```

1  #include "log.h"
2
3  void show_int_message(struct seq_file *m, const char *const f, const long
num) {
4      char tmp[256];

```

```

5     int len;
6
7     len = snprintf(tmp, 256, f, num);
8     seq_write(m, tmp, len);
9 }
10
11 void show_int3_message(struct seq_file *m, const char *const f, const long
    n1, const long n2, const long n3) {
12     char tmp[256];
13     int len;
14
15     len = snprintf(tmp, 256, f, n1, n2, n3);
16     seq_write(m, tmp, len);
17 }
18
19 void show_str_message(struct seq_file *m, const char *const f, const char *
    const s) {
20     char tmp[256];
21     int len;
22
23     len = snprintf(tmp, 256, f, s);
24     seq_write(m, tmp, len);
25 }

```

Листинг 18: листинг файла memory.h

```

1  #ifndef __MEMORY_H__
2  #define __MEMORY_H__
3
4  #include <linux/kthread.h>
5  #include <linux/delay.h>
6  #include <linux/time.h>
7
8  #include "log.h"
9
10 typedef struct mem_struct {
11     long available;
12     long free;
13     long time_secs;
14 } mem_info_t;
15

```

```

16  #define MEMORY_ARRAY_SIZE 8640
17  extern mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
18
19  extern int mem_info_calls_cnt;
20
21  #endif

```

Листинг 19: листинг файла hooks.c

```

1  #include "hooks.h"
2
3  #pragma GCC optimize("-fno-optimize-sibling-calls")
4
5  #if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,17,0)
6      )
7  #define PTREGS_SYSCALL_STUBS 1
8  #endif
9
10 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
11 #define FTRACE_OPS_FL_RECURSION FTRACE_OPS_FL_RECURSION_SAFE
12 #endif
13
14 #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
15 #define ftrace_regs pt_regs
16
17 static __always_inline struct pt_regs *ftrace_get_regs(struct ftrace_regs *
18     fregs)
19 {
20     return fregs;
21 }
22 #endif
23
24 ktime_t start_time;
25 static DEFINE_SPINLOCK(my_lock);
26
27 static void inline update_syscall_array(int syscall_num) {
28     ktime_t time;
29
30     time = ktime_get_boottime_seconds() - start_time;
31
32     spin_lock(&my_lock);

```

```

31
32     if (syscall_num < 64) {
33         syscalls_time_array[time % TIME_ARRAY_SIZE].p1 |= 1UL << syscall_num;
34     } else {
35         syscalls_time_array[time % TIME_ARRAY_SIZE].p2 |= 1UL << (syscall_num %
36             64);
37     }
38     spin_unlock(&my_lock);
39 }
40
41 /* 0 - sys_read */
42 #ifdef PTREGS_SYSCALL_STUBS
43 static asmlinkage long (*real_sys_read)(struct pt_regs *regs);
44
45 static asmlinkage long hook_sys_read(struct pt_regs *regs)
46 {
47     update_syscall_array(SYS_READ_NUM);
48     return real_sys_read(regs);
49 }
50 #else
51 static asmlinkage long (*real_sys_read)(unsigned int fd, char __user *buf,
52     size_t count);
53
54 static asmlinkage long hook_sys_read(unsigned int fd, char __user *buf,
55     size_t count)
56 {
57     update_syscall_array(SYS_READ_NUM);
58     return real_sys_read(fd, buf, count);
59 }
60 #endif
61
62 /* 1 - sys_write */
63 #ifdef PTREGS_SYSCALL_STUBS
64 static asmlinkage long (*real_sys_write)(struct pt_regs *regs);
65
66 static asmlinkage long hook_sys_write(struct pt_regs *regs)
67 {
68     update_syscall_array(SYS_WRITE_NUM);
69     return real_sys_write(regs);

```

```

68  }
69  #else
70  static asmlinkage long (*real_sys_write)(unsigned int fd, const char __user
    *buf, size_t count);
71
72  static asmlinkage long hook_sys_write(unsigned int fd, const char __user *
    buf, size_t count)
73  {
74      update_syscall_array(SYS_WRITE_NUM);
75      return real_sys_write(fd, buf, count);
76  }
77  #endif
78
79  /* 2 - sys_open */
80  #ifdef PTREGS_SYSCALL_STUBS
81  static asmlinkage long (*real_sys_open)(struct pt_regs *regs);
82
83  static asmlinkage long hook_sys_open(struct pt_regs *regs)
84  {
85      update_syscall_array(SYS_OPEN_NUM);
86      return real_sys_open(regs);
87  }
88  #else
89  static asmlinkage long (*real_sys_open)(const char __user *filename, int
    flags, umode_t mode);
90
91  static asmlinkage long hook_sys_open(const char __user *filename, int flags
    , umode_t mode);
92  {
93      update_syscall_array(SYS_OPEN_NUM);
94      return real_sys_open(filename, flags, mode);
95  }
96  #endif
97
98  /* 3 - sys_close */
99  #ifdef PTREGS_SYSCALL_STUBS
100  static asmlinkage long (*real_sys_close)(struct pt_regs *regs);
101
102  static asmlinkage long hook_sys_close(struct pt_regs *regs)
103  {

```

```

104     update_syscall_array(SYS_CLOSE_NUM);
105     return real_sys_close(regs);
106 }
107 #else
108 static asmlinkage long (*real_sys_close)(unsigned int fd);
109
110 static asmlinkage long hook_sys_close(unsigned int fd);
111 {
112     update_syscall_array(SYS_CLOSE_NUM);
113     return real_sys_close(fd);
114 }
115 #endif
116
117 /* 9 - sys_mmap */
118 #ifdef PTREGS_SYSCALL_STUBS
119 static asmlinkage long (*real_sys_mmap)(struct pt_regs *regs);
120
121 static asmlinkage long hook_sys_mmap(struct pt_regs *regs)
122 {
123     update_syscall_array(SYS_MMAP_NUM);
124     return real_sys_mmap(regs);
125 }
126 #else
127 static asmlinkage long (*real_sys_mmap)(unsigned int fd);
128
129 static asmlinkage long hook_sys_mmap(unsigned long addr, unsigned long len,
130 int prot, int flags,
131 int fd, long off)
132 {
133     update_syscall_array(SYS_CLOSE_NUM);
134     return real_sys_mmap(addr, len, prot, flags, fd, off);
135 }
136 #endif
137
138 /* 24 - sys_sched_yield */
139 #ifdef PTREGS_SYSCALL_STUBS
140 static asmlinkage long (*real_sys_sched_yield)(struct pt_regs *regs);
141
142 static asmlinkage long hook_sys_sched_yield(struct pt_regs *regs)
143 {

```



```

144     update_syscall_array(SYS_SCHED_YIELD_NUM);
145     return real_sys_sched_yield(regs);
146 }
147 #else
148 static asmlinkage long (*real_sys_sched_yield)(void);
149
150 static asmlinkage long hook_sys_sched_yield(void)
151 {
152     update_syscall_array(SYS_SCHED_YIELD_NUM);
153     return real_sys_sched_yield();
154 }
155 #endif
156
157 /* 41 - sys_socket */
158 #ifdef PTREGS_SYSCALL_STUBS
159 static asmlinkage long (*real_sys_socket)(struct pt_regs *regs);
160
161 static asmlinkage long hook_sys_socket(struct pt_regs *regs)
162 {
163     update_syscall_array(SYS_SOCKET_NUM);
164     return real_sys_socket(regs);
165 }
166 #else
167 static asmlinkage long (*real_sys_socket)(int, int, int);
168
169 static asmlinkage long hook_sys_socket(int a, int b, int c)
170 {
171     update_syscall_array(SYS_SOCKET_NUM);
172     return real_sys_socket(a, b, c);
173 }
174 #endif
175
176 /* 42 - sys_connect */
177 #ifdef PTREGS_SYSCALL_STUBS
178 static asmlinkage long (*real_sys_connect)(struct pt_regs *regs);
179
180 static asmlinkage long hook_sys_connect(struct pt_regs *regs)
181 {
182     update_syscall_array(SYS_CONNECT_NUM);
183     return real_sys_connect(regs);

```

```

184 }
185 #else
186 static asmlinkage long (*real_sys_connect)(int, struct sockaddr __user *,
187     int);
188
189 static asmlinkage long hook_sys_connect(int a, struct sockaddr __user * b,
190     int c);
191 {
192     update_syscall_array(SYS_CONNECT_NUM);
193     return real_sys_connect(a, b, c);
194 }
195 #endif
196
197 /* 43 - sys_accept */
198 #ifdef PTREGS_SYSCALL_STUBS
199 static asmlinkage long (*real_sys_accept)(struct pt_regs *regs);
200
201 static asmlinkage long hook_sys_accept(struct pt_regs *regs)
202 {
203     update_syscall_array(SYS_ACCEPT_NUM);
204     return real_sys_accept(regs);
205 }
206 #else
207 static asmlinkage long (*real_sys_accept)(int, struct sockaddr __user *,
208     int __user *);
209
210 static asmlinkage long hook_sys_accept(int a, struct sockaddr __user * b,
211     int __user *c)
212 {
213     update_syscall_array(SYS_ACCEPT_NUM);
214     return real_sys_accept(a, b, c);
215 }
216 #endif
217
218 /* 44 - sys_sendto */
219 #ifdef PTREGS_SYSCALL_STUBS
220 static asmlinkage long (*real_sys_sendto)(struct pt_regs *regs);
221
222 static asmlinkage long hook_sys_sendto(struct pt_regs *regs)
223 {

```

```

220     update_syscall_array(SYS_SENDDTO_NUM);
221     return real_sys_sendto(regs);
222 }
223 #else
224 static asmlinkage long (*real_sys_sendto)(int, void __user *, size_t,
225     unsigned,
226     struct sockaddr __user *, int);
227 static asmlinkage long hook_sys_sendto(int a, void __user * b, size_t c,
228     unsigned d,
229     struct sockaddr __user *e, int f);
230 {
231     update_syscall_array(SYS_SENDDTO_NUM);
232     return real_sys_sendto(a, b, c, d, e, f);
233 }
234 #endif
235
236 /* 45 - sys_recvfrom */
237 #ifdef PTREGS_SYSCALL_STUBS
238 static asmlinkage long (*real_sys_recvfrom)(struct pt_regs *regs);
239 static asmlinkage long hook_sys_recvfrom(struct pt_regs *regs)
240 {
241     update_syscall_array(SYS_RECVFROM_NUM);
242     return real_sys_recvfrom(regs);
243 }
244 #else
245 static asmlinkage long (*real_sys_recvfrom)(int, void __user *, size_t,
246     unsigned,
247     struct sockaddr __user *, int __user *)
248 static asmlinkage long hook_sys_recvfrom(int a, void __user *b, size_t c,
249     unsigned d,
250     struct sockaddr __user * e, int __user *f)
251 {
252     update_syscall_array(SYS_RECVFROM_NUM);
253     return real_sys_recvfrom(a, b, c, d, e, f);
254 }
255 #endif

```

```

256  /* 46 - sys_sendmsg */
257  #ifdef PTREGS_SYSCALL_STUBS
258  static asmlinkage long (*real_sys_sendmsg)(struct pt_regs *regs);
259
260  static asmlinkage long hook_sys_sendmsg(struct pt_regs *regs)
261  {
262      update_syscall_array(SYS_SENDMSG_NUM);
263      return real_sys_sendmsg(regs);
264  }
265  #else
266  static asmlinkage long (*real_sys_sendmsg)(int fd, struct user_msghdr
      __user *msg, unsigned flags);
267
268  static asmlinkage long hook_sys_sendmsg(int fd, struct user_msghdr __user *
      msg, unsigned flags)
269  {
270      update_syscall_array(SYS_SENDMSG_NUM);
271      return real_sys_sendmsg(fd, msg, flags);
272  }
273  #endif
274
275  /* 47 - sys_recvmsg */
276  #ifdef PTREGS_SYSCALL_STUBS
277  static asmlinkage long (*real_sys_recvmsg)(struct pt_regs *regs);
278
279  static asmlinkage long hook_sys_recvmsg(struct pt_regs *regs)
280  {
281      update_syscall_array(SYS_RECVMSG_NUM);
282      return real_sys_recvmsg(regs);
283  }
284  #else
285  static asmlinkage long (*real_sys_recvmsg)(int fd, struct user_msghdr
      __user *msg, unsigned flags);
286
287  static asmlinkage long hook_sys_recvmsg(int fd, struct user_msghdr __user *
      msg, unsigned flags)
288  {
289      update_syscall_array(SYS_RECVMSG_NUM);
290      return real_sys_recvmsg(fd, msg, flags);
291  }

```

```

292  #endif
293
294  /* 48 - sys_shutdown */
295  #ifdef PTREGS_SYSCALL_STUBS
296  static asmlinkage long (*real_sys_shutdown)(struct pt_regs *regs);
297
298  static asmlinkage long hook_sys_shutdown(struct pt_regs *regs)
299  {
300      update_syscall_array(SYS_SHUTDOWN_NUM);
301      return real_sys_shutdown(regs);
302  }
303  #else
304  static asmlinkage long (*real_sys_shutdown)(int, int);
305
306  static asmlinkage long hook_sys_shutdown(int t, int m)
307  {
308      update_syscall_array(SYS_SHUTDOWN_NUM);
309      return real_sys_shutdown(t, m);
310  }
311  #endif
312
313  /* 56 - sys_clone */
314  #ifdef PTREGS_SYSCALL_STUBS
315  static asmlinkage long (*real_sys_clone)(struct pt_regs *regs);
316
317  static asmlinkage long hook_sys_clone(struct pt_regs *regs)
318  {
319      update_syscall_array(SYS_CLONE_NUM);
320      return real_sys_clone(regs);
321  }
322  #else
323  static asmlinkage long (*real_sys_clone)(unsigned long clone_flags,
324  unsigned long newsp, int __user *parent_tidptr,
325  int __user *child_tidptr, unsigned long tls);
326
327  static asmlinkage long hook_sys_clone(unsigned long clone_flags,
328  unsigned long newsp, int __user *parent_tidptr,
329  int __user *child_tidptr, unsigned long tls)
330  {
331      update_syscall_array(SYS_CLONE_NUM);

```

```

332     return real_sys_clone(clone_flags, newsp, parent_tidptr, child_tidptr,
        tls);
333 }
334 #endif
335
336 /* 59 - sys_execve */
337 #ifdef PTREGS_SYSCALL_STUBS
338 static asmlinkage long (*real_sys_execve)(struct pt_regs *regs);
339
340 static asmlinkage long hook_sys_execve(struct pt_regs *regs)
341 {
342     update_syscall_array(SYS_EXECVE_NUM);
343     return real_sys_execve(regs);
344 }
345 #else
346 static asmlinkage long (*real_sys_execve)(const char __user *filename,
347     const char __user *const __user *argv,
348     const char __user *const __user *envp);
349
350 static asmlinkage long hook_sys_execve(const char __user *filename,
351     const char __user *const __user *argv,
352     const char __user *const __user *envp)
353 {
354     update_syscall_array(SYS_EXECVE_NUM);
355     return real_sys_execve(filename, argv, envp);
356 }
357 #endif
358
359 /* 83 - sys_mkdir */
360 #ifdef PTREGS_SYSCALL_STUBS
361 static asmlinkage long (*real_sys_mkdir)(struct pt_regs *regs);
362
363 static asmlinkage long hook_sys_mkdir(struct pt_regs *regs)
364 {
365     update_syscall_array(SYS_MKDIR_NUM);
366     return real_sys_mkdir(regs);
367 }
368 #else
369 static asmlinkage long (*real_sys_mkdir)(const char __user *pathname,
        umode_t mode);

```

```

370
371 static asmlinkage long hook_sys_mkdir(const char __user *pathname, umode_t
    mode);
372 {
373     update_syscall_array(SYS_MKDIR_NUM);
374     return real_sys_mkdir(pathname, mode);
375 }
376 #endif
377
378 /* 84 - sys_rmdir */
379 #ifdef PTREGS_SYSCALL_STUBS
380 static asmlinkage long (*real_sys_rmdir)(struct pt_regs *regs);
381
382 static asmlinkage long hook_sys_rmdir(struct pt_regs *regs)
383 {
384     update_syscall_array(SYS_RMDIR_NUM);
385     return real_sys_rmdir(regs);
386 }
387 #else
388 static asmlinkage long (*real_sys_rmdir)(const char __user *pathname);
389
390 static asmlinkage long hook_sys_rmdir(const char __user *pathname);
391 {
392     update_syscall_array(SYS_RMDIR_NUM);
393     return real_sys_rmdir(pathname);
394 }
395 #endif
396
397
398 /*
399  * x86_64 kernels have a special naming convention for syscall entry points
    in newer kernels.
400  * That's what you end up with if an architecture has 3 (three) ABIs for
    system calls.
401  */
402 #ifdef PTREGS_SYSCALL_STUBS
403 #define SYSCALL_NAME(name) ("__x64_" name)
404 #else
405 #define SYSCALL_NAME(name) (name)
406 #endif

```

```

407
408 #define ADD_HOOK(_name, _function, _original) \
409 { \
410     .name = SYSCALL_NAME(_name), \
411     .function = (_function), \
412     .original = (_original), \
413 }
414
415 static struct ftrace_hook hooked_functions[] = {
416     ADD_HOOK("sys_execve", hook_sys_execve, &real_sys_execve),
417     ADD_HOOK("sys_write", hook_sys_write, &real_sys_write),
418     ADD_HOOK("sys_open", hook_sys_open, &real_sys_open),
419     ADD_HOOK("sys_close", hook_sys_close, &real_sys_close),
420     ADD_HOOK("sys_mmap", hook_sys_mmap, &real_sys_mmap),
421     ADD_HOOK("sys_sched_yield", hook_sys_sched_yield, &real_sys_sched_yield
),
422     ADD_HOOK("sys_socket", hook_sys_socket, &real_sys_socket),
423     ADD_HOOK("sys_connect", hook_sys_connect, &real_sys_connect),
424     ADD_HOOK("sys_accept", hook_sys_accept, &real_sys_accept),
425     ADD_HOOK("sys_sendto", hook_sys_sendto, &real_sys_sendto),
426     ADD_HOOK("sys_recvfrom", hook_sys_recvfrom, &real_sys_recvfrom),
427     ADD_HOOK("sys_sendmsg", hook_sys_sendmsg, &real_sys_sendmsg),
428     ADD_HOOK("sys_recvmsg", hook_sys_recvmsg, &real_sys_recvmsg),
429     ADD_HOOK("sys_shutdown", hook_sys_shutdown, &real_sys_shutdown),
430     ADD_HOOK("sys_read", hook_sys_read, &real_sys_read),
431     ADD_HOOK("sys_clone", hook_sys_clone, &real_sys_clone),
432     ADD_HOOK("sys_mkdir", hook_sys_mkdir, &real_sys_mkdir),
433     ADD_HOOK("sys_rmdir", hook_sys_rmdir, &real_sys_rmdir),
434 };
435
436 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
437 static unsigned long lookup_name(const char *name)
438 {
439     struct kprobe kp = {
440         .symbol_name = name
441     };
442     unsigned long retval;
443
444     ENTER_LOG();
445

```



```

446     if (register_kprobe(&kp) < 0) {
447         EXIT_LOG();
448         return 0;
449     }
450
451     retval = (unsigned long) kp.addr;
452     unregister_kprobe(&kp);
453
454     EXIT_LOG();
455
456     return retval;
457 }
458 #else
459 static unsigned long lookup_name(const char *name)
460 {
461     unsigned long retval;
462
463     ENTER_LOG();
464     retval = kallsyms_lookup_name(name);
465     EXIT_LOG();
466
467     return retval;
468 }
469 #endif
470
471 static int resolve_hook_address(struct ftrace_hook *hook)
472 {
473     ENTER_LOG();
474
475     if (!(hook->address = lookup_name(hook->name))) {
476         pr_debug("unresolved symbol: %s\n", hook->name);
477         EXIT_LOG();
478         return -ENOENT;
479     }
480
481     *((unsigned long*) hook->original) = hook->address;
482
483     EXIT_LOG();
484
485     return 0;

```

```

486 }
487
488 static void notrace ftrace_thunk(unsigned long ip, unsigned long parent_ip,
489 struct ftrace_ops *ops, struct ftrace_regs *fregs)
490 {
491     struct pt_regs *regs = ftrace_get_regs(fregs);
492     struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
493
494     if (!within_module(parent_ip, THIS_MODULE)) {
495         regs->ip = (unsigned long)hook->function;
496     }
497 }
498
499 static int install_hook(struct ftrace_hook *hook) {
500     int rc;
501
502     ENTER_LOG();
503
504     if ((rc = resolve_hook_address(hook))) {
505         EXIT_LOG();
506         return rc;
507     }
508
509     /* Callback function. */
510     hook->ops.func = ftrace_thunk;
511     /* Save processor registers. */
512     hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
513         | FTRACE_OPS_FL_RECURSION
514         | FTRACE_OPS_FL_IPMODIFY;
515
516     /* Turn of ftrace for our function. */
517     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0))) {
518         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
519         return rc;
520     }
521
522     /* Allow ftrace call our callback. */
523     if ((rc = register_ftrace_function(&hook->ops))) {
524         pr_debug("register_ftrace_function() failed: %d\n", rc);
525         ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);

```

```

526     }
527
528     EXIT_LOG();
529
530     return rc;
531 }
532
533 static void remove_hook(struct ftrace_hook *hook) {
534     int rc;
535
536     ENTER_LOG();
537
538     if (hook->address == 0x00) {
539         EXIT_LOG();
540         return;
541     }
542
543     if ((rc = unregister_ftrace_function(&hook->ops))) {
544         pr_debug("unregister_ftrace_function() failed: %d\n", rc);
545     }
546
547     if ((rc = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0))) {
548         pr_debug("ftrace_set_filter_ip() failed: %d\n", rc);
549     }
550
551     hook->address = 0x00;
552
553     EXIT_LOG();
554 }
555
556 int install_hooks(void) {
557     size_t i;
558     int rc;
559
560     ENTER_LOG();
561
562     for (i = 0; i < ARRAY_SIZE(hooked_functions); i++) {
563         if ((rc = install_hook(&hooked_functions[i]))) {
564             pr_debug("instal_hooks failed: %d\n", rc);
565             goto err;

```

```

566     }
567 }
568
569 EXIT_LOG();
570
571 return 0;
572
573 err:
574 while (i != 0) {
575     remove_hook(&hooked_functions[--i]);
576 }
577
578 EXIT_LOG();
579
580 return rc;
581 }
582
583 void remove_hooks(void) {
584     size_t i;
585
586     ENTER_LOG();
587
588     for (i = 0; i < ARRAY_SIZE(hooked_functions); i++) {
589         remove_hook(&hooked_functions[i]);
590     }
591
592     EXIT_LOG();
593 }

```

Листинг 20: листинг файла hooks.h

```

1  #ifndef __HOOKS_H_
2  #define __HOOKS_H_
3
4  #include <linux/kprobes.h>
5  #include <linux/version.h>
6  #include <linux/ftrace.h>
7  #include <linux/time.h>
8
9  #include "log.h"
10

```

```

11  #define SYS_READ_NUM 0
12  #define SYS_WRITE_NUM 1
13  #define SYS_OPEN_NUM 2
14  #define SYS_CLOSE_NUM 3
15
16  #define SYS_MMAP_NUM 9
17
18  #define SYS_SCHED_YIELD_NUM 24
19
20  #define SYS_SOCKET_NUM 41
21  #define SYS_CONNECT_NUM 42
22  #define SYS_ACCEPT_NUM 43
23  #define SYS_SENDTO_NUM 44
24  #define SYS_RECVFROM_NUM 45
25  #define SYS_SENDMSG_NUM 46
26  #define SYS_RECVMSG_NUM 47
27  #define SYS_SHUTDOWN_NUM 48
28
29  #define SYS_CLONE_NUM 56
30  #define SYS_EXECVE_NUM 59
31
32  #define SYS_MKDIR_NUM 83
33  #define SYS_RMDIR_NUM 84
34
35  struct ftrace_hook {
36      const char *name;
37      void *function;
38      void *original;
39
40      unsigned long address;
41      struct ftrace_ops ops;
42  };
43
44  typedef struct {
45      uint64_t p1;
46      uint64_t p2;
47  } syscalls_info_t;
48
49  #define TIME_ARRAY_SIZE 86400
50  extern syscalls_info_t syscalls_time_array[TIME_ARRAY_SIZE];

```

```
51
52  void remove_hooks(void);
53  int install_hooks(void);
54
55  #endif
```