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 _____ А. А. Петрова
(Группа) (Подпись, дата) (И.О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

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 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на 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:

- выбрать способ, наиболее отвечающий поставленному заданию;
- разработать алгоритмы и структуру ПО;
- разработать ПО;
- проанализировать результаты работы разработанного ПО.

Требования к разрабатываемому ПО:

- ПО должно выводить на экран терминала:
  - информацию об оперативной памяти за определенный промежуток времени;
  - количество системных вызовов;
- полученная информация должна записываться в виртуальную файловую систему /proc в файлы /proc/monitor/memory и /proc/monitor/syscalls.

1.2 Информация об оперативной памяти

1.2.1 Структура struct sysinfo

Структура struct sysinfo [4] 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    __kernel_ulong_t totalhigh; /* Total high memory size */
12    __kernel_ulong_t freehigh;  /* Available high memory size */
13    __u32 mem_unit;     /* Memory unit size in bytes */
14    char _f[20-2*sizeof(__kernel_ulong_t)-sizeof(int)]; /* Padding to 64
      bytes */
15  };
```

Наиболее важными полями для данной работы являются: totalram (общий объем используемой оперативной памяти), freeram (объем свободной оперативной памяти), sharedram (объем разделяемой памяти), bufferram (память, используемая буферами), totalswap (общий размер swap пространства), totalhigh (общий объем верхней области памяти (HMA, память, зарезервированная для системного аппаратного обеспечения)), freehigh (объем свободной HMA).

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().

При этом в качестве промежутка времени, через который фиксируется состояние оперативной памяти, выбрана 1 секунда. Данный выбор обусловлен тем, что интервал больше одной секунды является слишком широким для сбора такой статистики, в то время как информация, приходящая чаще, чем через секунду, будет сложна для анализа из-за ее объема.

1.3 Количество системных вызовов

Для подсчета количества системных вызовов необходимо перехватывать эти системные вызовы.

Перехват функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции таким образом, чтобы при вызове этой функции управление передавалось не ей, а функции, которая будет её подменять. Эта функция, работая вместо системной, выполняет какие-то запланированные действия (в данном случае увеличивает счетчик количества вызовов перехваченной функции), и затем, либо вызывает оригинальный обработчик системного вызова, либо не вызывает его вообще.

Далее будут рассмотрены различные существующие подходы к перехвату вызываемых функций и выбран наиболее подходящий для реализации в данной работе.

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 Linux Security Modules

Linux Security Modules (LSM) [5] – это специальный интерфейс, созданный для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые вызывают коллбеки (англ. callback), установленные security-модулем. Данный модуль может изучать контекст операции и принимать решение о её разрешении или запрете.

Особенности рассматриваемого интерфейса:

- security-модули являются частью ядра и не могут быть загружены динамически;
- в стандартной конфигурации сборки ядра флаг наличия LSM неактивен – большинство уже готовых сборок ядра не содержат внутри себя интерфейс LSM;
- в системе может быть только один security-модуль.

Таким образом, для использования Linux Security Modules необходимо поставлять собственную сборку ядра Linux, что является трудоёмким вариантом – как минимум, придётся тратить время на сборку ядра. Кроме того, данный интерфейс обладает излишним функционалом (например решение о блокировке какой-либо операции), который не потребуется в написании разрабатываемого модуля ядра.

1.3.3 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. Таким 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 оста-

нова, 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).

1.3.4 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].

1.3.5 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 конфигура-

циях установлены все необходимые флаги для работы;

1.3.6 Сравнительный анализ методов

В таблице 1 приведено сравнение рассмотренных методов.

Таблица 1 – Методы перехвата системных вызовов

Название	Дин. загрузка	Перехват любых функций	Любая конфигурация ядра	Простота реализации	Информация об адресе перехватываемой функции	Наличие документации
Модификация таблицы системных вызовов	+	–	+	–	+	–
Linux Security Module	–	+	–	–	–	–
kprobes	+	+	+	–	+	+
kernel tracepoints	+	+	–	+	–	–
ftrace	+	+	–	+	–	+

По результатам сравнения для перехвата системных вызовов был выбран ftrace, так как он удовлетворяет большинству важных требований (возможность динамической загрузки, перехвата системных вызовов, простота реализации и

наличие документации). При этом для получения информации об адресе перехватываемой функции можно использовать kprobes.

Выводы

В результате проведенного анализа для реализации перехвата системных вызовов был выбран фреймворк ftrace, так как он не требует специальной сборки ядра и предоставляет возможность динамической загрузки в ядро. При этом для поиска адреса перехватываемой функции был выбран интерфейс kprobes, так как ftrace не предоставляет такой возможности.

Помимо этого были рассмотрены структура struct sysinfo и функция si_mem_available, предоставляющие информацию об объеме доступной и занятой оперативной памяти, объеме разделяемой памяти, размере swap пространства и т. д. При этом в качестве промежутка времени, через который фиксируется информация об оперативной памяти, была выбрана 1 секунда.

2 Конструкторский раздел

2.1 Алгоритм получения информации об объеме доступной и занятой оперативной памяти

На рисунке 1 представлена схема алгоритма работы потока ядра, предназначенного для подсчета объема свободной и занятой оперативной памяти в системе.

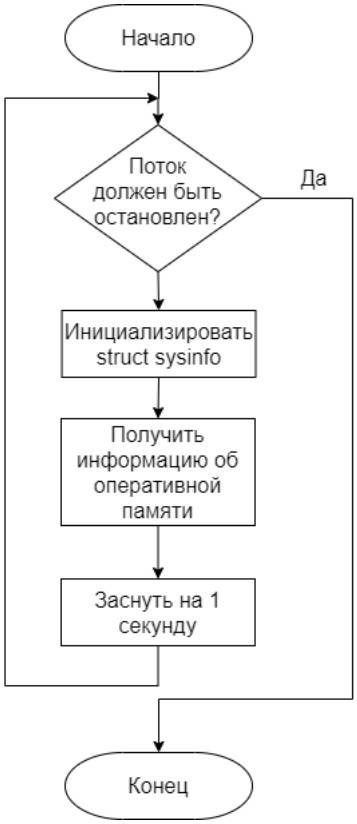

Рисунок 1 – Схема алгоритма работы потока

Поток ядра находится в состоянии сна и, просыпаясь каждые 10 секунд, фиксирует информацию о свободной и занятой оперативной памяти.

2.2 Алгоритм перехвата системного вызова

На рисунках 2-3 представлен алгоритм перехвата системных вызовов на примере sys_clone.

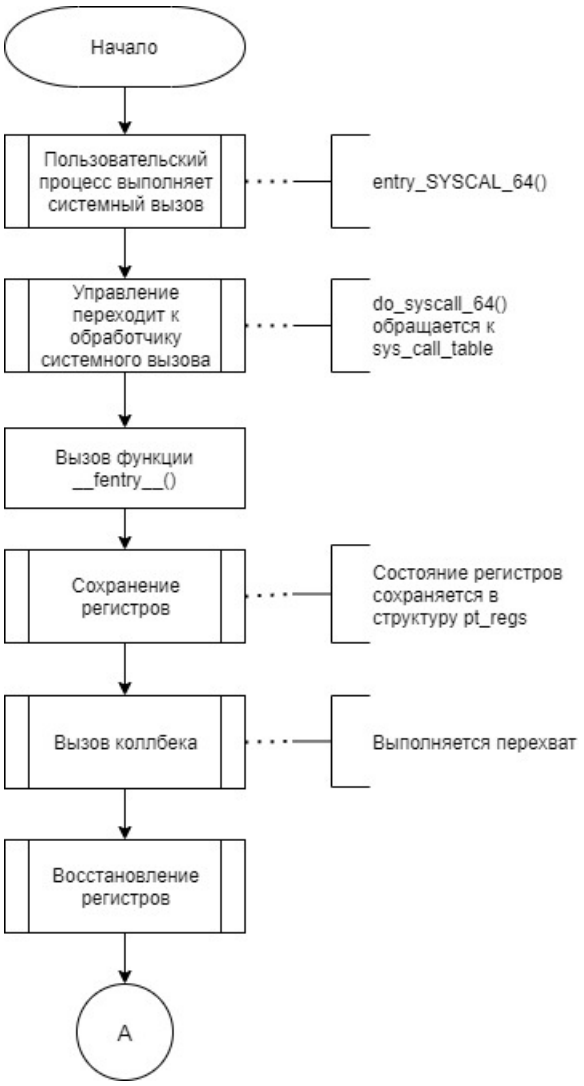

Рисунок 2 – Алгоритм перехвата системного вызова (ч. 1)

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 функ-

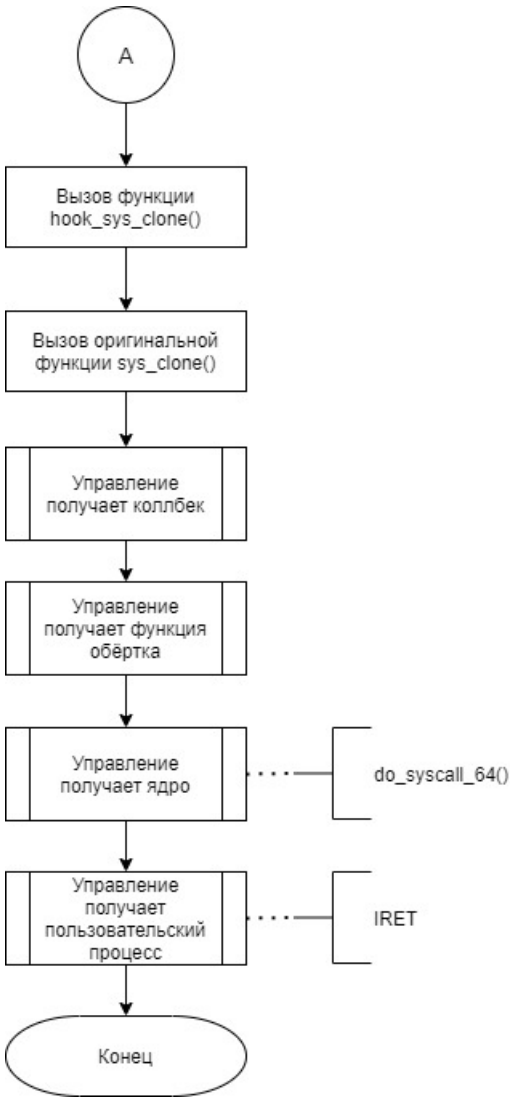

Рисунок 3 – Алгоритм перехвата системного вызова (ч. 2)

ции __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 регистр rip – это приведёт к передачу управления по новому адресу.
7) Управление получает функция обёртка. Благодаря безусловному перехо-

ду, 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 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 4 представлена схема алгоритма подсчёта системных вызовов.

Рисунок 4 – 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 вызов 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 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.

2.5 Точки входа в загружаемый модуль

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

При инициализации модуля создаются необходимые для сохранения результатов директория и файлы в /proc, создается и запускается поток ядра для

подсчета объема свободной и занятой оперативной памяти и устанавливаются 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.

2.6 Структура ПО

На рисунке 5 представлена структура разрабатываемого программного обеспечения.

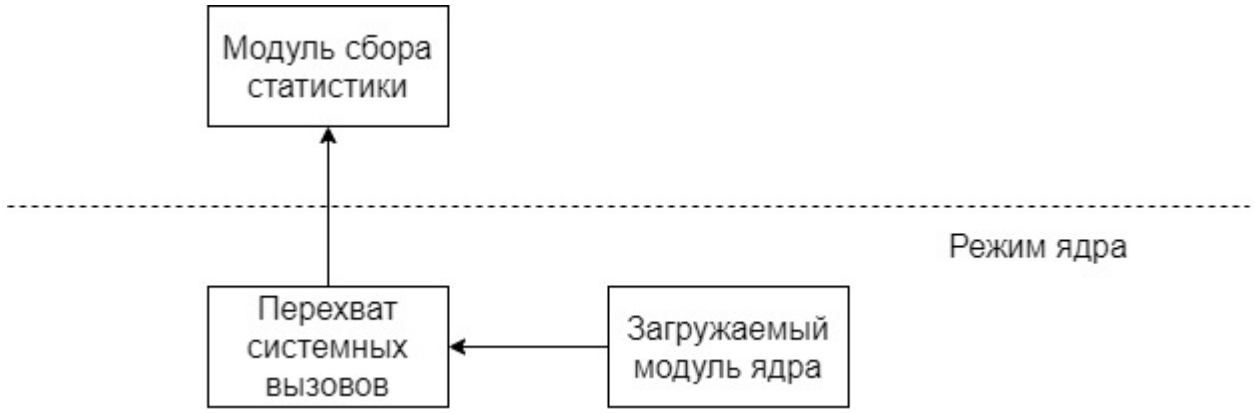

Рисунок 5 – Структура программного обеспечения

3 Технологический раздел

3.1 Выбор языка и среды 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], а в качестве среды программирования – QtCreator.

3.2 Информация о памяти в системе

Для сбора информации об оперативной памяти в системе запускается отдельный поток ядра, который находится в состоянии сна и, просыпаясь каждую секунду, 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].shared = i.sharedram;
20     mem_info_array[mem_info_calls_cnt].buffers = i.bufferram;
21     mem_info_array[mem_info_calls_cnt].swap = i.totalswap;
```

```
22        mem_info_array[mem_info_calls_cnt].totalhigh = i.totalhigh;
23        mem_info_array[mem_info_calls_cnt].freehigh = i.freehigh;
24        mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
25
26        ssleep(1);
27
28        if (signal_pending(worker_task)) {
29          break;
30        }
31      }
32
33      EXIT_LOG();
34      do_exit(0);
35      return 0;
36    }
```

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.

Начиная с версии ядра 5.7.0 функция kallsyms_lookup_name() [14], позволяющая найти адрес перехватываемой функции, перестала быть экспортируемой. Так как модуль ядра разрабатывался на системе с версией ядра 5.15.0, для поиска адреса перехватываемой функции использовался интерфейс kprobes.

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 мас-

сив, 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,
     ktime_t range) {
12    int syscalls_arr_cnt[128];
13    uint64_t tmp;
14    size_t i;
15    ktime_t uptime;
16
17    memset((void*)syscalls_arr_cnt, 0, 128 * sizeof(int));
18    uptime = ktime_get_boottime_seconds() - mstart;
19
20    if (uptime < range) {
21      range = uptime;
22    }
23
24    for (i = 0; i < range; i++) {
25      if ((tmp = syscalls_time_array[uptime - i].p1) != 0) {
26        walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt);
27      }
28
29      if ((tmp = syscalls_time_array[uptime - i].p2) != 0) {
30        walk_bits_and_find_syscalls(m, tmp, syscalls_arr_cnt + 64);
31      }
32    }
33
34    show_int_message(m, "Syscall statistics for the last %d seconds.\n\n",
     range);
35
36    for (i = 0; i < 128; i++) {
37      if (syscalls_arr_cnt[i] != 0) {
38        show_str_message(m, "%s called ", syscalls_names[i]);
39        show_int_message(m, "%d times.\n", syscalls_arr_cnt[i]);
40      }
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 Исследовательский раздел

4.1 Результаты работы разработанного ПО

На рисунках 6-7 представлены примеры работы модуля (статистика по количеству свободной и занятой оперативной памяти и по количеству системных вызовов).

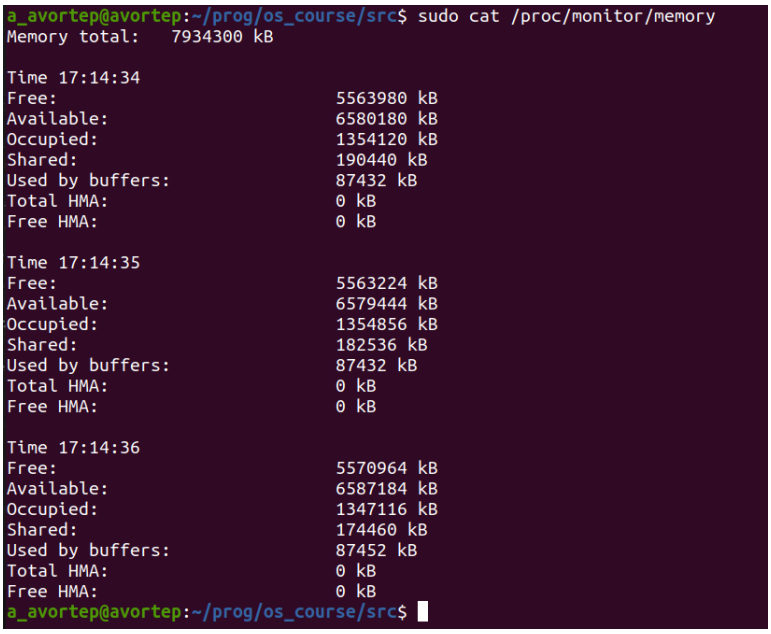

Рисунок 6 – Информация об оперативной памяти в системе

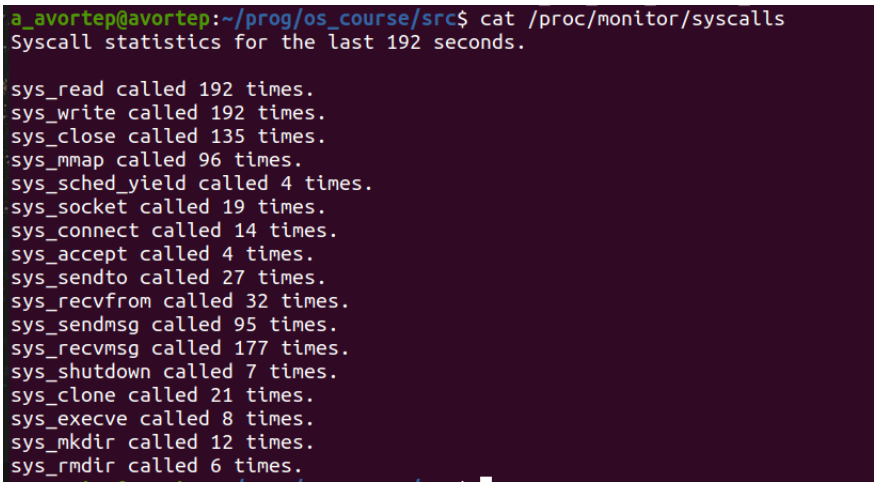

Рисунок 7 – Информация о количестве системных вызовов за последние 192 секунды

На рисунке 8 представлено сравнение результатов получения информации об объеме доступной и занятой оперативной памяти с результатами команды «free -m» (у последней все объемы указаны в Мб).

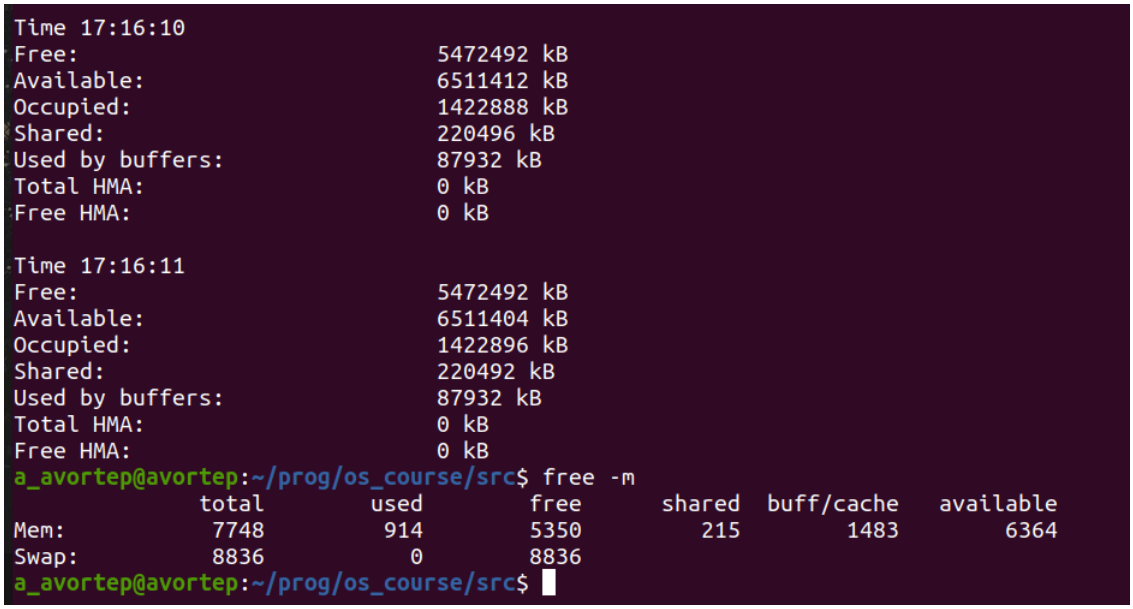

Рисунок 8 – Результаты работы команды «free -m»

Как видно из скриншота 8, результаты работы разработанного загружаемого модуля ядра близки к результатам работы команды «free -m».

На рисунке 9 представлена визуализация данных о свободной, доступной и занятой памяти в системе, полученных из разработанного модуля ядра.

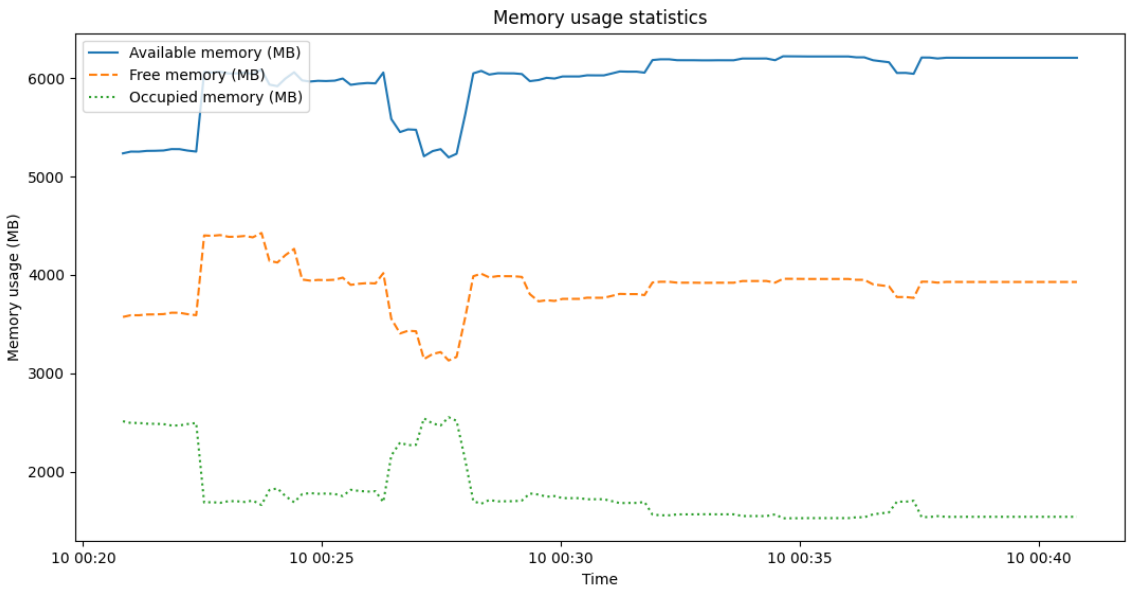

Рисунок 9 – Визуализация данных о памяти за 20 минут

По графикам 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 загруженности 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. 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).

5. Linux Security Modules: General Security Hooks for Linux [Электронный ресурс]. – Режим доступа: https://docs.kernel.org/security/lsm.html (дата обращения: 13.02.2023).

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    return single_open(sp_file, show_syscalls, NULL);
38  }
39
40  static int proc_release(struct inode *sp_node, struct file *sp_file) {
41    return 0;
42  }
43
44  mem_info_t mem_info_array[MEMORY_ARRAY_SIZE];
45  int mem_info_calls_cnt;
46
47  int memory_cnt_task_handler_fn(void *args) {
48    struct sysinfo i;
49    struct timespec64 t;
50
51    ENTER_LOG();
52
53    allow_signal(SIGKILL);
54
55    while (!kthread_should_stop()) {
56      si_meminfo(&i);
57
58      ktime_get_real_ts64(&t);
59
60      mem_info_array[mem_info_calls_cnt].free = i.freeram;
61      mem_info_array[mem_info_calls_cnt].available = si_mem_available();
62      mem_info_array[mem_info_calls_cnt++].time_secs = t.tv_sec;
63
64      ssleep(10);
65
66      if (signal_pending(worker_task)) {
67        break;
68      }
69    }
70
71    EXIT_LOG();
72    do_exit(0);
73    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: xxhyymzzs. 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, &
        syscalls_ops)) == NULL)
154     {
155       goto err;
156     }
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
        counter thread");
186     kthread_bind(worker_task, cpu);
187
188     if (worker_task == NULL) {
189       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     "sysgettimeofday",
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_getpgid",
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   #define PTREGS_SYSCALL_STUBS 1
7   #endif
8
9   #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
10  #define FTRACE_OPS_FL_RECURSION FTRACE_OPS_FL_RECURSION_SAFE
11  #endif
12
13  #if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
14  #define ftrace_regs pt_regs
15
16  static __always_inline struct pt_regs *ftrace_get_regs(struct ftrace_regs *
     fregs)
17  {
18    return fregs;
19  }
20  #endif
21
22  ktime_t start_time;
23  static DEFINE_SPINLOCK(my_lock);
24
25  static void inline update_syscall_array(int syscall_num) {
26    ktime_t time;
27
28    time = ktime_get_boottime_seconds() - start_time;
29
30    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 %
        64);
36      }
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,
       size_t count);
52
53    static asmlinkage long hook_sys_read(unsigned int fd, char __user *buf,
       size_t count)
54    {
55      update_syscall_array(SYS_READ_NUM);
56      return real_sys_read(fd, buf, count);
57    }
58    #endif
59
60    /* 1 - sys_write */
61    #ifdef PTREGS_SYSCALL_STUBS
62    static asmlinkage long (*real_sys_write)(struct pt_regs *regs);
63
64    static asmlinkage long hook_sys_write(struct pt_regs *regs)
65    {
66      update_syscall_array(SYS_WRITE_NUM);
67      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 *,
       int);
187
188   static asmlinkage long hook_sys_connect(int a, struct sockaddr __user * b,
       int c);
189   {
190     update_syscall_array(SYS_CONNECT_NUM);
191     return real_sys_connect(a, b, c);
192   }
193   #endif
194
195   /* 43 - sys_accept */
196   #ifdef PTREGS_SYSCALL_STUBS
197   static asmlinkage long (*real_sys_accept)(struct pt_regs *regs);
198
199   static asmlinkage long hook_sys_accept(struct pt_regs *regs)
200   {
201     update_syscall_array(SYS_ACCEPT_NUM);
202     return real_sys_accept(regs);
203   }
204   #else
205   static asmlinkage long (*real_sys_accept)(int, struct sockaddr __user *,
       int __user *)
206
207   static asmlinkage long hook_sys_accept(int a, struct sockaddr __user * b,
       int __user *c)
208   {
209     update_syscall_array(SYS_ACCEPT_NUM);
210     return real_sys_accept(a, b, c);
211   }
212   #endif
213
214   /* 44 - sys_sendto */
215   #ifdef PTREGS_SYSCALL_STUBS
216   static asmlinkage long (*real_sys_sendto)(struct pt_regs *regs);
217
218   static asmlinkage long hook_sys_sendto(struct pt_regs *regs)
219   {
```

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

```
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
```