

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Постановка задачи	4
1.2 Перехват функций	4
1.2.1 Linux Security Module	5
1.2.2 Модификация таблицы системных вызовов	5
1.2.3 kprobes	6
1.2.4 Kernel tracepoints	7
1.2.5 ftrace	7
1.3 Информация о процессах и памяти	9
1.3.1 Структура <code>struct task_struct</code>	9
1.3.2 Структура <code>struct sysinfo</code>	11
1.4 Загружаемые модули ядра	12
1.4.1 Пространство ядра и пользователя	12
1.5 Виртуальная файловая система <code>/proc</code>	13
2 Конструкторская часть	16
2.1 Архитектура приложения	16
2.2 Структура <code>struct ftrace_hook</code>	16
2.3 Алгоритм перехвата системного вызова	17
2.4 Алгоритм подсчёта количества системных вызовов	20
3 Технологическая часть	22
3.1 Выбор языка программирования	22
3.2 Поиск адреса перехватываемой функции	22
3.3 Инициализация <code>ftrace</code>	23
3.4 Функции обёртки	25
3.5 Получение информации о количестве системных вызовов	26
3.6 Информация о памяти в системе	27
3.7 Получение информации о процессах	28
3.8 Примеры работы разработанного ПО	30

Заключение	33
Литература	34

Введение

В настоящее время большую актуальность имеют системы, предоставляющие информацию о ресурсах операционной системы и частоте системных вызовов. Предоставив такую информацию, пользователь сможет 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.

Данная работа посвящена исследованию структур ядра, хранящим информацию о процессах в системе и памяти, способам перехвата системных вызовов ядра с их последующим логированием.

Целью данной курсовой работы является разработка загружаемого модуля ядра, предоставляющего информацию о количестве системных вызовов и выделенной памяти за выбранный промежуток времени и информацию о состоянии всех процессов в системе в текущий момент.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с заданием необходимо разработать загружаемый модуль ядра, который позволит посмотреть количество системных вызовов, свободной и доступной оперативной памяти за выбранный промежуток времени, а так же количество процессов и их состояния на данный момент.

Для решение данной задачи необходимо:

- проанализировать различные подходы к перехвату функций;
- исследовать структуры и функции ядра, предоставляющие информацию о процессах и памяти;
- изучить методы передачи информации из пространства ядра в пространство пользователя и наоборот;
- спроектировать и реализовать загружаемый модулей ядра.

1.2 Перехват функций

Перехват функции заключается в изменении некоторого адреса в памяти процесса или кода в теле функции таким образом, чтобы при вызове этой самой функции управление передавалось не ей, а функции, которая будет её подменять. Эта функция, работая вместо системной, выполняет какие-то запланированные действия, и затем, либо вызывает оригинальную функцию, либо не вызывает ее вообще.

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

1.2.1 Linux Security Module

Linux Security Module (LSM) [2] – это специальный интерфейс, созданный для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые вызывают коллбеки (англ. callback [3]), установленные security-модулем. Данный модуль может изучать контекст операции и принимать решение о её разрешении или запрете [2].

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 [2].

Таким образом, для использования Linux Security Module 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.

1.2.2 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 накладные расходы;

- не требуется специальная конфигурация ядра;
- техническая сложность реализации – необходимо модифицировать таблицу системных вызовов;
- из-за ряда оптимизаций, реализованных в ядре, некоторые обработчики невозможно перехватить [4];
- можно перехватить только системные вызовы – нельзя перехватить обычные функции.

1.2.3 kprobes

kprobes [5] – специальный интерфейс, предназначенный для отладки и трассировки ядра. Данный интерфейс позволяет устанавливать пред- и пост-обработчики для любой инструкции в ядре, а так же обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут изменять их значение. Таким образом, **kprobes** можно использовать как и в целях мониторинга, так и для возможности повлиять на дальнейший ход работы ядра [4].

Особенности рассматриваемого интерфейса:

- перехват любой инструкции в ядре – это реализуется с помощью точек останова (инструкция `int3`), внедряемых в исполняемый код ядра. Таким образом, можно перехватить любую функцию в ядре;
- хорошо задокументированный API;
- нетривиальные накладные расходы – для расстановки и обработки точек останова необходимо большое количество процессорного времени [4];
- техническая сложность реализации. Так, например, чтобы получить аргументы функции или значения её локальных переменных нужно знать, в каких регистрах, или в каком месте на стеке они находятся, и самостоятельно их оттуда извлекать;

- при подмене адреса возврата из функции используется стек, реализованный с помощью буфера фиксированного размера. Таким образом, при большом количестве одновременных вызовов перехваченной функции, могут быть пропущены срабатывания.

1.2.4 Kernel tracepoints

`Kernel tracepoints` [6] – это фреймворк для трассировки ядра, реализованный через статическое инструментирование кода. Большинство важных функций ядра статически инструментировано – в теле функций добавлены вызовы функций фреймворка рассматриваемого фреймворка.

Особенности рассматриваемого фреймворка:

- минимальные накладные расходы – необходимо только вызвать функцию трассировки в необходимом месте;
- отсутствие задокументированного API;
- не все функции ядра статически инструментированны;
- не работает, если ядро не сконфигурировано должным образом [4].

1.2.5 ftrace

`ftrace` [7] – это фреймворк для трассировки ядра на уровне функций, реализованный на основе ключей компилятора `-pg` [8] и `mentry` [8]. Данные функции вставляют в начало каждой функции вызов специальной трассировочной функции `mcount()` или `__fentry()__`. В пользовательских программах данная возможность компилятора используется профилировщиками, с целью отслеживания всех вызываемых функций. В ядре эти функции используются исключительно для реализации рассматриваемого фреймворка.

Для большинства современных архитектур процессора доступна оптимизация: динамический `frace` [8]. Ядро знает расположение всех вызовов функций `mcount()` или `__fentry()__` и на ранних этапах загрузки ядра

подменяет их машинный код на специальную машинную инструкцию NOP [9], которая ничего не делает. При включении трассировки, в нужные функции необходимые вызовы добавляются обратно. Если **ftrace** не используется, его влияние на производительность системы минимально.

Особенности рассматриваемого фреймворка:

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

Сравнение методов

В таблице 1.1 приведено сравнение приведенных выше методов, позволяющих перехватывать системные вызовы.

Название	Дин. загрузка	Перехват любых функций	Любая конфигурация ядра	Простота реализации	Наличие документации
Linux Security Module	Нет	Да	Нет	Нет	Нет
Модификация таблицы системных вызовов	Да	Нет	Да	Нет	Нет
kprobes	Да	Да	Да	Нет	Да
kernel tracepoints	Да	Да	Нет	Да	Нет
ftrace	Да	Да	Нет	Да	Да

Таблица 1.1: Методы перехвата системных вызовов

1.3 Информация о процессах и памяти

1.3.1 Структура `struct task_struct`

Информация о процессах в ядре хранится с помощью специальной структуры `struct task_struct` [10]. Каждому процессу в системе соответствует структура `task_struct`, которая полностью описывает процесс. Сами структуры связаны друг с другом по средствам кольцевого связанного списка.

Структура описывает текущее состояние процесса, его флаги, указатель на процессы-потомки и так далее. Стоит отметить, что для описания потоков, в ядре Linux так же используется данная структура – различие лишь в установленных флагах. В листинге 1.1 представлено объявление структуры с наиболее интересными полями.

```
1 struct task_struct {
2     #ifdef CONFIG_THREAD_INFO_IN_TASK
3     struct thread_info    thread_info;
4     #endif
5
6     unsigned int          __state;
7     ...
8     unsigned int          flags;
9     ...
10    #ifdef CONFIG_SMP
11    int                    on_cpu;
12    ...
13    int                    recent_used_cpu;
14    #endif
15    ...
16    int                    recent_used_cpu;
17    ...
18    #ifdef CONFIG_CGROUP_SCHED
19    struct task_group      *sched_task_group;
20    #endif
21    ...
22    struct sched_info      sched_info;
23    ...
24    struct list_head        tasks;
25    ...
26 }
```

Листинг 1.1: Листинг структуры `task_struct` с наиболее интересными полями

Для работы с данной структурой внутри ядра объявлен ряд макросов. Например, чтобы обойти все процессы в системе, существует макрос `for_each_process`, который итерируется по связанному списку процессов. Состояния процесса так же описываются с помощью специальных макросов. Кроме того, существует ряд макросов, позволяющих проверить текущее состояние процесса, например, узнать, выполняется ли процесс в данный момент. Список этих макросов приведён в листингах 1.2 - 1.3.

```
1 #define TASK_RUNNING      0x0000
2 #define TASK_INTERRUPTIBLE 0x0001
3 #define TASK_UNINTERRUPTIBLE 0x0002
4 #define __TASK_STOPPED    0x0004
5 #define __TASK_TRACED     0x0008
6 #define EXIT_DEAD        0x0010
7 #define EXIT_ZOMBIE      0x0020
8 #define EXIT_TRACE       (EXIT_ZOMBIE | EXIT_DEAD)
9 #define TASK_PARKED      0x0040
10 #define TASK_DEAD        0x0080
11 #define TASK_WAKEKILL    0x0100
12 #define TASK_WAKING      0x0200
13 #define TASK_NOLOAD      0x0400
14 #define TASK_NEW         0x0800
15 #define TASK_RTLOCK_WAIT 0x1000
16 #define TASK_STATE_MAX   0x2000
17 #define TASK_KILLABLE     (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
18 #define TASK_STOPPED      (TASK_WAKEKILL | __TASK_STOPPED)
19 #define TASK_TRACED       (TASK_WAKEKILL | __TASK_TRACED)
20 #define TASK_IDLE         (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)
21 #define TASK_NORMAL       (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
22 #define TASK_REPORT       (TASK_RUNNING | TASK_INTERRUPTIBLE | \
23 TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
24 __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
25 TASK_PARKED)
```

Листинг 1.2: Описание состояний процесса с помощью макросов

```
1 #define task_is_running(task)  (READ_ONCE((task)->__state) == TASK_RUNNING)
2 #define task_is_traced(task)   ((READ_ONCE(task->__state) & __TASK_TRACED) !=
3                                0)
4 #define task_is_stopped(task)  ((READ_ONCE(task->__state) & __TASK_STOPPED)
5                                != 0)
```

```
4 #define task_is_stopped_or_traced(task) ((READ_ONCE(task->__state) & (
    __TASK_STOPPED | __TASK_TRACED)) != 0)
```

Листинг 1.3: Макросы с помощью которых можно узнать текущее состояние процесса

1.3.2 Структура `struct sysinfo`

Структура `struct sysinfo` [11] хранит информацию статистику о всей системе: информацию о времени, прошедшем с начала запуска системы, количество занятой памяти и так далее. В листинге 1.4 приведёно объявление рассматриваемой структуры.

```
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 sharedram; /* Amount of shared memory */
6     __kernel_ulong_t bufferram; /* Memory used by buffers */
7     __kernel_ulong_t totalswap; /* Total swap space size */
8     __kernel_ulong_t freeswap; /* swap space still available */
9     __u16 procs;               /* Number of current processes */
10    __u16 pad;                  /* Explicit padding for m68k */
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(__u32)];
15 };
```

Листинг 1.4: Листинг структуры `struct sysinfo`

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

1.4 Загружаемые модули ядра

Одной из особенностей ядра Linux является способность расширения функциональности во время работы, без необходимости компиляции ядра заново. Таким образом, существует возможность добавить (или убрать) функциональность в ядро можно когда система запущена и работает. Часть кода, которая может быть добавлена в ядро во время работы, называется модулем ядра. Ядро Linux предлагает поддержку большого числа классов модулей. Каждый модуль – это подготовленный объектный код, который может быть динамически подключен в работающее ядро, а позднее может быть выгружен из ядра.

Каждый модуль ядра сам регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Задача инициализации модуля заключается в подготовке функций модуля для последующего вызова. Функция выхода модуля вызывается перед выгрузкой модуля из ядра. Функция выхода должна отменить все изменения, сделанные функцией инициализации, освободить захваченные в процессе работы модуля ресурсы.

Возможность выгрузить модуль помогает сократить время разработки – нет необходимости перезагрузки компьютера при последовательном тестировании новых версий разрабатываемого модуля ядра.

Модуль связан только с ядром и может вызывать только те функции, которые экспортированы ядром.

1.4.1 Пространство ядра и пользователя

Приложения работают в пользовательском пространстве, а ядро и его модули – в пространстве ядра. Такое разделение пространств – базовая концепция теории операционных систем.

Ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих задач становится возмож-

ным только в том случае, если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли и некоторые операции на более низких уровнях не допускаются; программный код может переключить один уровень на другой только ограниченным числом способов. Все современные процессоры имеют не менее двух уровней защиты, а некоторые, например семейство процессоров x86, имеют больше уровней; когда существует несколько уровней, используется самый высокий и самый низкий уровень защиты.

Ядро Linux выполняется на самом высоком уровне, где разрешено выполнение любых инструкций и доступ к произвольным участкам памяти, а приложения выполняются на самом низком уровне, в котором процессор регулирует прямой доступ оборудованию и несанкционированный доступ к памяти. Ядро выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, выполняя системный вызов, работает в контексте процесса – он действует от имени вызывающего процесса и в состоянии получить данные в адресном пространстве процесса. Код, который обрабатывает прерывания является асинхронным по отношению к процессам и не связан с каким-либо определенным процессом [4].

Ролью модуля ядра является расширение функциональности ядра без его перекомпиляции. Код модулей выполняется в пространстве ядра.

1.5 Виртуальная файловая система /proc

Для организации доступа к разнообразным файловым системам в Unix используется промежуточный слой абстракции – виртуальная файловая система. С точки зрения программиста, виртуальная файловая система организована как специальный интерфейс. Виртуальная файловая система объявляет API доступа к ней, а реализацию этого API отдаёт на откуп к драйверам конкретных файловых систем.

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

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

Ядро предоставляет возможность добавить своё дерево в каталог `/proc`. Внутри ядра объявлена специальная структура `struct proc_ops` [12]. Эта структура содержит внутри себя указатели на функции чтения файла, записи в файл и прочие, определенные пользователем. В листинге [?] представлено объявление данной структуры в ядре.

```
1 struct proc_ops {
2     unsigned int proc_flags;
3     int (*proc_open)(struct inode *, struct file *);
4     ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
6     ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
7     loff_t (*proc_lseek)(struct file *, loff_t, int);
8     int (*proc_release)(struct inode *, struct file *);
9     __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
10    long (*proc_ioctl)(struct file *, unsigned int, unsigned long);
11    #ifdef CONFIG_COMPAT
12    long (*proc_compat_ioctl)(struct file *, unsigned int, unsigned long);
13    #endif
14    int (*proc_mmap)(struct file *, struct vm_area_struct *);
15    unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
16    unsigned long, unsigned long, unsigned long);
16 } __randomize_layout;
```

Листинг 1.5: Листинг структуры `struct sysinfo`

С помощью вызова функций `proc_mkdir()` и `proc_create()` в модуле ядра можно зарегистрировать свои каталоги и файлы в `/proc` соответственно. Функции `copy_to_user()` и `copy_from_user()` реализуют передачу данных (набора байтов) из пространства ядра в пространство пользователя

и наоборот.

Таким образом, с помощью виртуальной файловой системы `/proc` можно получать (или передавать) какую-либо информацию из пространства ядра в пространство пользователя (из пространства пользователя в пространство ядра).

Вывод

В данном разделе были проанализированы различные подходы к перехвату функций. В ходе анализа, 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 и имеет хорошо задокументированный API. Были рассмотрены структуры и функции ядра, предоставляющие информацию о процессах и памяти; рассмотрены особенности загружаемых модулей ядра и понятия пространств ядра и пространства пользователя, а так же рассмотрен способ взаимодействия этих двух пространств с целью передачи данных из одного в другого.

2 Конструкторская часть

2.1 Архитектура приложения

В состав разработанного программного обеспечения входит один загружаемый модуль ядра, который перехватывает все вызовы системных вызовов, подсчитывая их количество за определенный промежуток времени, предоставляет пользователю информацию о процессах и их состояниях, а так же информацию и состояние о загруженности оперативной памяти – её общее количество, свободной и доступной в данный момент.

2.2 Структура `struct ftrace_hook`

В листинге 2.1 представлено объявление структуры `struct ftrace_hook`, которая описывает каждую перехватываемую функцию.

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

Листинг 2.1: Листинг структуры `ftrace_hook`

Необходимо заполнить только первые три поля:

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

Остальные поля считаются деталью реализации. Описание всех перехватываемых были собраны в массив, а для инициализации был написан

специальный макрос (см. листинг 2.2).

```
1 #define ADD_HOOK(_name, _function, _original) \  
2 { \  
3     .name = SYSCALL_NAME(_name), \  
4     .function = (_function), \  
5     .original = (_original), \  
6 } \  
7 \  
8 static struct ftrace_hook hooked_functions[] = { \  
9     ADD_HOOK("sys_execve", hook_sys_execve, &real_sys_execve), \  
10    ADD_HOOK("sys_write", hook_sys_write, &real_sys_write), \  
11    ADD_HOOK("sys_open", hook_sys_open, &real_sys_open), \  
12    ADD_HOOK("sys_close", hook_sys_close, &real_sys_close), \  
13    ADD_HOOK("sys_mmap", hook_sys_mmap, &real_sys_mmap), \  
14    ADD_HOOK("sys_sched_yield", hook_sys_sched_yield, &real_sys_sched_yield), \  
15    ADD_HOOK("sys_socket", hook_sys_socket, &real_sys_socket), \  
16    ADD_HOOK("sys_connect", hook_sys_connect, &real_sys_connect), \  
17    ADD_HOOK("sys_accept", hook_sys_accept, &real_sys_accept), \  
18    ADD_HOOK("sys_sendto", hook_sys_sendto, &real_sys_sendto), \  
19    ADD_HOOK("sys_recvfrom", hook_sys_recvfrom, &real_sys_recvfrom), \  
20    ADD_HOOK("sys_sendmsg", hook_sys_sendmsg, &real_sys_sendmsg), \  
21    ADD_HOOK("sys_recvmsg", hook_sys_recvmsg, &real_sys_recvmsg), \  
22    ADD_HOOK("sys_shutdown", hook_sys_shutdown, &real_sys_shutdown), \  
23    ADD_HOOK("sys_read", hook_sys_read, &real_sys_read), \  
24    ADD_HOOK("sys_clone", hook_sys_clone, &real_sys_clone), \  
25    ADD_HOOK("sys_mkdir", hook_sys_mkdir, &real_sys_mkdir), \  
26    ADD_HOOK("sys_rmdir", hook_sys_rmdir, &real_sys_rmdir), \  
27 };
```

Листинг 2.2: Объявление массива перехватываемых функций и специальный макрос для его инициализации

2.3 Алгоритм перехвата системного вызова

На рисунке 2.1 представлена схема алгоритма перехвата системных вызовов на примере `sys_clone`.

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

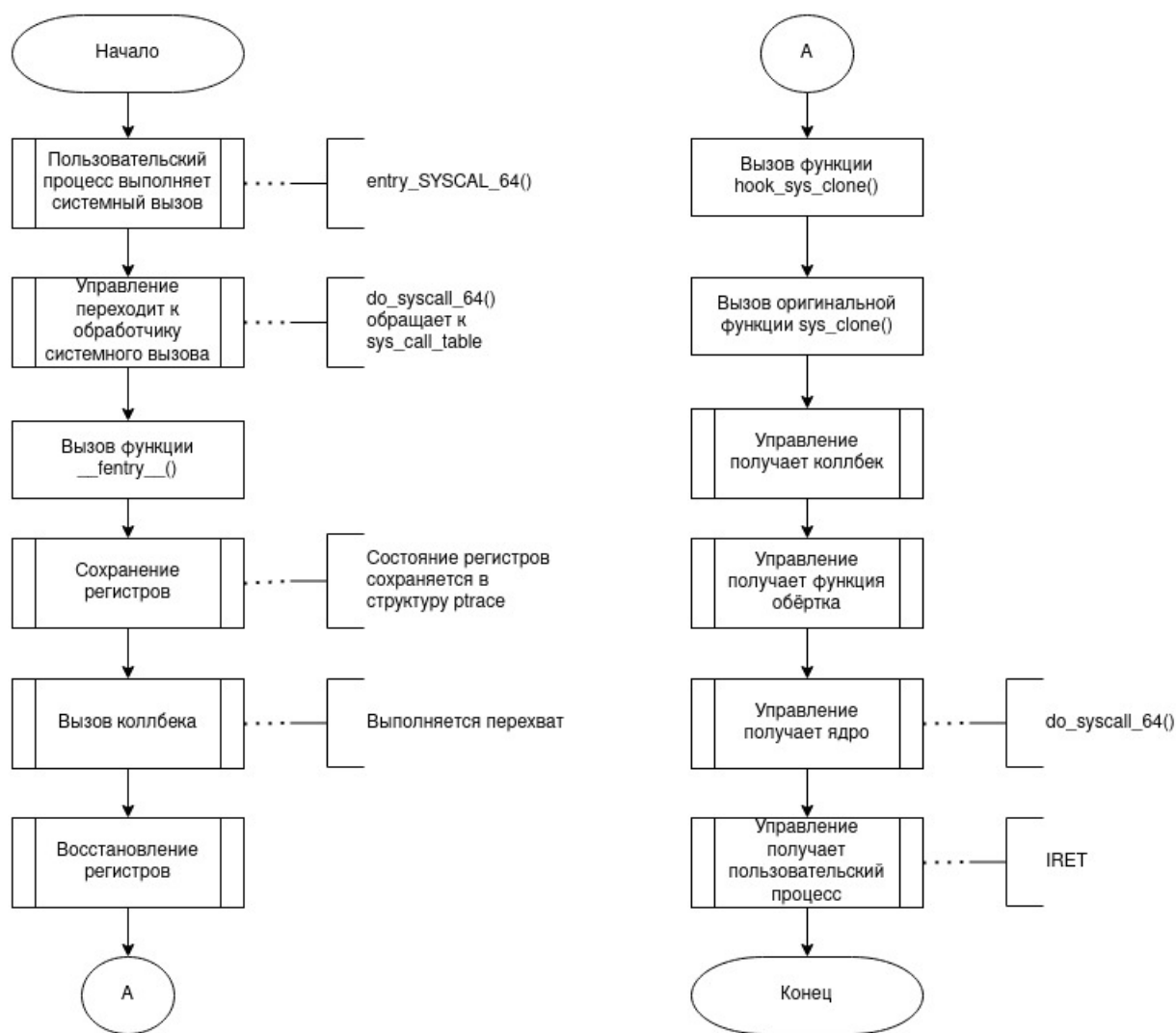


Рис. 2.1: Алгоритм перехвата системного вызова

`entry_SYSCALL_64()`. Этот обработчик отвечает за все системные вызовы 64-битных программ на 64-битных машинах.

- 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()`.
- 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 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.

2.4 Алгоритм подсчёта количества системных вызовов

На рисунке 2.2 представлена схема алгоритма подсчёта системных вызовов.



Рис. 2.2: Алгоритм подсчёта количества системных вызовов

- Агрегирующий массив – это массив на 86400 элементов, состоящий из структур, имеющих два поля в виде 64-битных без знаковых 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;

Вывод

В данном разделе была рассмотрена общая архитектура приложения, алгоритм перехвата системных вызовов и подсчёта их количества за выбранный промежуток.

3 Технологическая часть

3.1 Выбор языка программирования

Разработанный модуль ядра написан на языке программирования C [13]. Выбор языка программирования C основан на том, что исходный код ядра Linux, все его модули и драйверы написаны на данном языке.

В качестве компилятора выбран gcc [14].

3.2 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 [15]) 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 5.14.9, воспользоваться данным способом было нельзя. В конечном итоге проблемы была решена с помощью интерфейса `kprobes` (который был описан в 1.1.3).

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()`. Такая реализация стала возможной благодаря директивам условной компиляции [16] и специальным макросам `LINUX_VERSION_CODE` и `KERNEL_VERSION()`.

Реализация функции `lookup_name()`, возвращающей адрес функции перехватываемой функции по её названию, представлена в листинге 3.1.

```
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

```

Листинг 3.1: Реализация функции `lookup_name()`

3.3 Инициализация `ftrace`

В листинге 3.2 представлена реализация функции, которая инициализирует структуру `ftrace_ops`.

```

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 }

```

Листинг 3.2: Реализация функции `install_hook()`

В листинге 3.3 представлена реализация отключения перехвата функции.

```

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();

```


Листинг 3.3: Реализация функции `remove_hook()`

3.4 Функции обёртки

При объявлении функций обёрток, которые будут запущены вместо перехватываемой функции, необходимо в точности соблюдать сигнатуру. Так, должны совпадать порядок, типы аргументов и возвращаемого значения. Оригинальные описания функций были из исходных кодов ядра Linux.

В листинге 3.4 представлена реализация функции обёртки на примере `sys_clone()`.

```

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 }
```

Листинг 3.4: Реализация функции обёртки

В листинге 3.5 представлена реализация функции которая обновляет массив, хранящий количество системных вызовов за последние 24 часа.

```

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 %
14         64);
15 }
16 spin_unlock(&my_lock);
17 }

```

Листинг 3.5: Реализация функции `update_syscall_array()`

3.5 Получение информации о количестве системных вызовов

В листинге 3.6 представлена реализация функций, которые агрегируют информацию о системных вызовах (данные массива `update_syscall_array`) и предоставляют ее в читаемом для пользователя виде.

```

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 }

```

Листинг 3.6: Реализация функций агрегации данных о системных вызовах

3.6 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 3.7 представлена реализация этого потока, а в листинге 3.8 его инициализация.

```

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 3.7: Реализация функции сохраняющей информацию о доступной в системе памяти

```

1 int my_thread_init() {
2     cpu = get_cpu();
3     worker_task = kthread_create(memory_cnt_task_handler_fn, NULL, "memory
4         counter thread");
5     kthread_bind(worker_task, cpu);
6
7     if (worker_task == NULL) {
8         cleanup();
9         return -1;
10    }
11
12    wake_up_process(worker_task);
13    return 0;
14 }

```

Листинг 3.8: Функция инициализации потока ядра

3.7 Получение информации о процессах

В листинге 3.9 представлена реализация функции, которая выводит информацию о состоянии всех текущих процессах на данный момент.

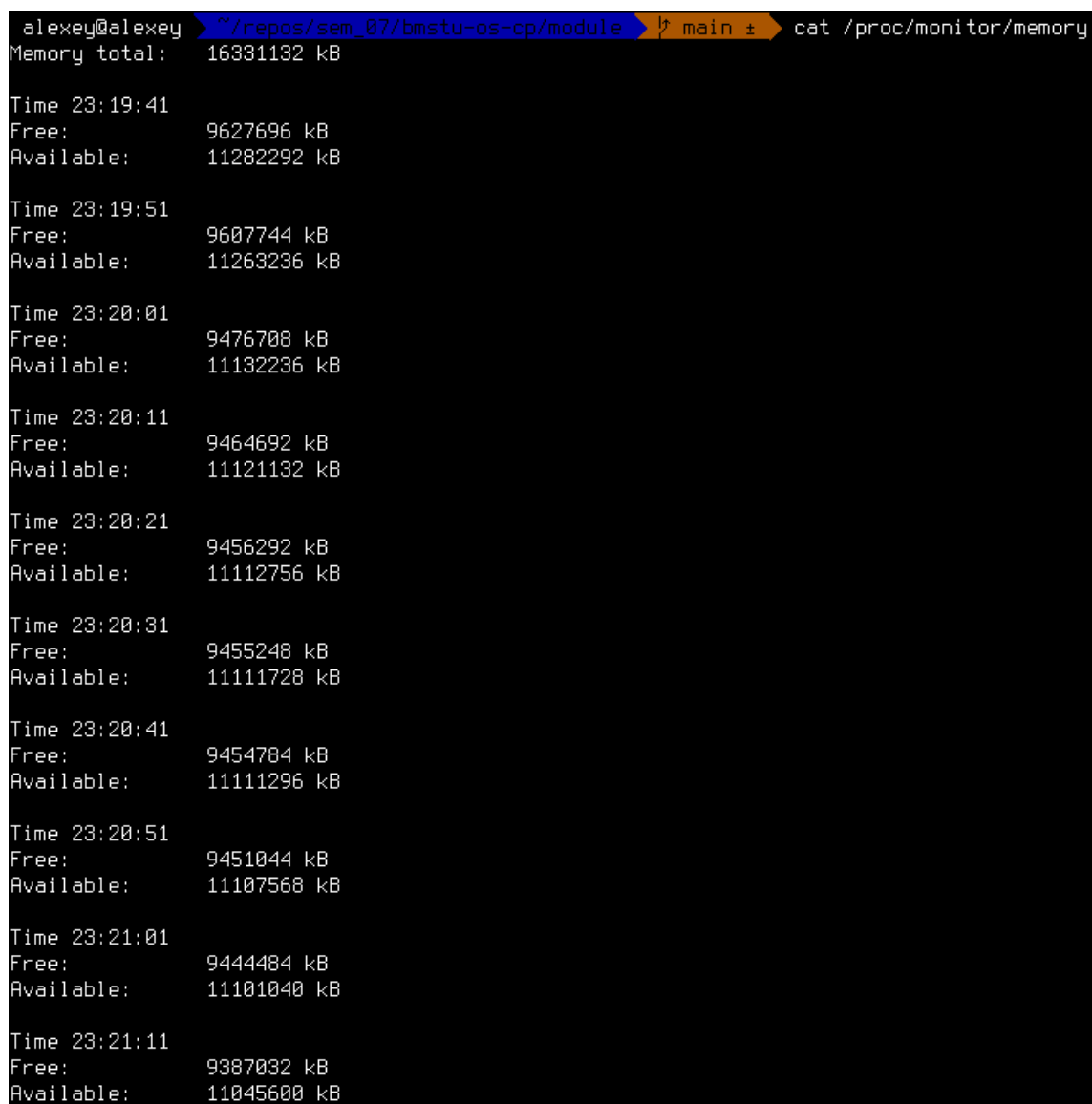
```

1 void print_task_statistics(struct seq_file *m) {
2     struct task_struct *task;
3     int total = 0, running = 0, stopped = 0, zombie = 0, interruptible = 0,
        uninterruptible;
4
5     ENTER_LOG();
6
7     for_each_process(task) {
8         switch (task->TASK_STATE_FIELD) {
9             case TASK_RUNNING:
10                running++;
11                break;
12             case TASK_INTERRUPTIBLE:
13                interruptible++;
14                break;
15             case TASK_IDLE: /* (TASK_UNINTERRUPTIBLE | TASK_NOLOAD) */
16                uninterruptible++;
17                break;
18             case TASK_STOPPED:
19                stopped++;
20                break;
21             case TASK_TRACED: /* (TASK_WAKEKILL | __TASK_TRACED) */
22                stopped++;
23                break;
24             default:
25                printk(KERN_INFO "%x %s %d\n", task->TASK_STATE_FIELD, task->comm, task
->pid);
26            }
27
28            if (task->exit_state == EXIT_ZOMBIE)
29            {
30                zombie++;
31            }
32
33            total++;
34        }
35
36        show_int_message(m, "Total processes: %d\n", total);
37        show_int_message(m, "Running: %d\n", running);
38        show_int_message(m, "Sleeping: %d ", total - running - stopped - zombie);
39        show_int_message(m, "[Interruptible: %d | ", interruptible);
40        show_int_message(m, "Uninterruptible: %d]\n", uninterruptible);
41        show_int_message(m, "Stopped: %d\n", stopped);
42        show_int_message(m, "Zombie: %d\n", zombie);
43
44        EXIT_LOG();
45    }

```

3.8 Примеры работы разработанного ПО

На рисунках 3.1 - 3.4 представлены примеры работы разработанного модуля ядра. Для наглядности перехватываются только 18 системных вызовов.



The screenshot shows a terminal window with a dark background. At the top, the prompt is 'alexey@alexey' followed by a blue bar containing '~/.repos/sem_07/bmstu-os-cp/module' and an orange bar containing 'main'. The command 'cat /proc/monitor/memory' has been executed. The output consists of several lines of memory statistics, each preceded by a timestamp. The statistics include 'Memory total', 'Free', and 'Available' memory in kB.

```
alexey@alexey > ~/.repos/sem_07/bmstu-os-cp/module main > cat /proc/monitor/memory
Memory total: 16331132 kB

Time 23:19:41
Free: 9627696 kB
Available: 11282292 kB

Time 23:19:51
Free: 9607744 kB
Available: 11263236 kB

Time 23:20:01
Free: 9476708 kB
Available: 11132236 kB

Time 23:20:11
Free: 9464692 kB
Available: 11121132 kB

Time 23:20:21
Free: 9456292 kB
Available: 11112756 kB

Time 23:20:31
Free: 9455248 kB
Available: 11111728 kB

Time 23:20:41
Free: 9454784 kB
Available: 11111296 kB

Time 23:20:51
Free: 9451044 kB
Available: 11107568 kB

Time 23:21:01
Free: 9444484 kB
Available: 11101040 kB

Time 23:21:11
Free: 9387032 kB
Available: 11045600 kB
```

Рис. 3.1: Информация о оперативной памяти в системе

```

alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 306
Running: 1
Sleeping: 303 [Interruptible: 203 | Uninterruptible: 100]
Stopped: 1
Zombie: 1
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 304
Running: 1
Sleeping: 302 [Interruptible: 202 | Uninterruptible: 100]
Stopped: 1
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 304
Running: 1
Sleeping: 302 [Interruptible: 202 | Uninterruptible: 100]
Stopped: 1
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/tasks
Total processes: 302
Running: 1
Sleeping: 301 [Interruptible: 201 | Uninterruptible: 100]
Stopped: 0
Zombie: 0
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ±

```

Рис. 3.2: Информация о процессах и их состояниях на текущий момент в системе

```

alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ± cat /proc/monitor/syscalls
Syscall statistics for the last 122 seconds.

sys_read called 122 times.
sys_write called 121 times.
sys_open called 2 times.
sys_close called 87 times.
sys_mmap called 69 times.
sys_sched_yield called 53 times.
sys_socket called 17 times.
sys_connect called 15 times.
sys_accept called 4 times.
sys_sendto called 86 times.
sys_recvfrom called 24 times.
sys_sendmsg called 111 times.
sys_recvmsg called 122 times.
sys_shutdown called 4 times.
sys_clone called 30 times.
sys_execve called 24 times.
sys_mkdir called 4 times.
sys_rmdir called 2 times.
alexey@alexey > ~/repos/sem_07/bmstu-os-cp/module > main ±

```

Рис. 3.3: Информация о количестве системных вызовов за последние 122 секунды

Вывод

В данном разделе был обоснован выбор языка программирования, рассмотрены листинги реализованного программного обеспечения и приведе-

```
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ± echo 00h00m15s > /proc/monitor/syscalls
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ± cat /proc/monitor/syscalls
Syscall statistics for the last 15 seconds.
sys_read called 15 times.
sys_write called 15 times.
sys_close called 14 times.
sys_mmap called 14 times.
sys_sched_yield called 1 times.
sys_socket called 1 times.
sys_connect called 1 times.
sys_sendto called 6 times.
sys_recvfrom called 4 times.
sys_sendmsg called 15 times.
sys_recvmsg called 15 times.
sys_clone called 3 times.
sys_execve called 2 times.
alexey@alexey ~/repos/sem_07/bmstu-os-cp/module main ±
```

Рис. 3.4: Конфигурирование модуля для отображение информации о системных вызовах за последние 15 секунд

ны результаты работы ПО.

Заключение

В ходе проделанной работы был разработан загружаемый модуль ядра, предоставляющий информацию о загрузке системы: количество системных вызовов за выбранный промежуток времени, количество свободной и доступной оперативной памяти, статистика по процессам и в каких состояниях они находятся.

Изучены структуры и функции ядра, которые предоставляют информацию о процессах и памяти. Проанализированы существующие подходы к перехвату системных вызовов.

На основе полученных знаний и проанализированных технологий реализован загружаемый модуль ядра.

Литература

- [1] Linux - Operating System [Электронный ресурс]. Режим доступа: <https://www.linux.org/> (дата обращения: 08.11.2021).
- [2] Linux Security Module Usage [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html> (дата обращения: 08.11.2021).
- [3] Колбэк-функция – Глоссарий – MDN Web Docs [Электронный ресурс]. Режим доступа: https://developer.mozilla.org/ru/docs/Glossary/Callback_function (дата обращения: 08.11.2021).
- [4] Механизмы профилирования Linux – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/metrotek/blog/261003/> (дата обращения: 08.11.2021).
- [5] Kernel Probes (Kprobes) [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/kprobes.html> (дата обращения: 08.11.2021).
- [6] Using the Linux Kernel Tracepoints [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/trace/tracepoints.html> (дата обращения: 08.11.2021).
- [7] Using ftrace | Android Open Source Project [Электронный ресурс]. Режим доступа: <https://source.android.com/devices/tech/debug/ftrace> (дата обращения: 08.11.2021).
- [8] Трассировка ядра с ftrace – Habr [Электронный ресурс]. Режим доступа: <https://habr.com/ru/company/selectel/blog/280322/> (дата обращения: 08.11.2021).
- [9] NOP: No Operation (x86 Instruction Set Reference) [Электронный ресурс]. Режим доступа: https://c9x.me/x86/html/file_module_x86_id_217.html (дата обращения: 08.11.2021).

- [10] `include/linux/sched.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h> (дата обращения: 08.11.2021).
- [11] `include/uapi/linux/sysinfo.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/sysinfo.h#L8> (дата обращения: 08.11.2021).
- [12] `include/linux/proc_fs.h` - Linux source code (v5.15.3) [Электронный ресурс]. Режим доступа: https://elixir.bootlin.com/linux/latest/source/include/linux/proc_fs.h#L29 (дата обращения: 08.11.2021).
- [13] C99 standard note [Электронный ресурс]. Режим доступа: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> (дата обращения: 10.11.2021).
- [14] GCC, the GNU Compiler Collection [Электронный ресурс]. Режим доступа: <https://gcc.gnu.org/> (дата обращения: 10.11.2021).
- [15] Unexporting `kallsyms_lookup_name()` [Электронный ресурс]. Режим доступа: <https://lwn.net/Articles/813350/> (дата обращения: 10.11.2021).
- [16] Директивы препроцессора C [Электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/preprocessor-directives> (дата обращения: 10.11.2021).