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 КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Загружаемый модуль ядра для мониторинга запускаемых приложений путем перехвата системного вызова exec»

Студент _____ИУ7-71Б_____ _____ _____Е.А.Панафидин_____
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ _____Н.Ю. Рязанова_____
(Подпись, дата) (И.О.Фамилия)

2021 г.

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)

УТВЕРЖДАЮ
Заведующий кафедрой ____ИУ7___
(Индекс)
_____ __И. В. Рудаков__
(И.О.Фамилия)
« _____ » _____ 20 ____ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине _____Операционные системы_____

Студент группы _____ИУ7-71Б_____
_____Панафидин Егор Алексеевич_____
(Фамилия, имя, отчество)

Тема курсового проекта _____Загружаемый модуль ядра для мониторинга запускаемых приложений путем перехвата системного вызова exec_____

Направленность КП (учебный, исследовательский, практический, производственный, др.)
_____учебный_____
_____
Источник тематики (кафедра, предприятие, НИР) _инициативная тема_____

График выполнения проекта: 25% к _4_ нед., 50% к _7_ нед., 75% к _11 нед., 100% к _14 нед.

1. Техническое задание

Разработать загружаемый модуль ядра для мониторинга запускаемых приложений путем перехвата системного вызова exec. Предусмотреть вывод информации о предке и количестве потомков, вызвавших exec.

2. ***Оформление курсового проекта:***

2.1 Расчетно-пояснительная записка на _25-30_ листах формата А4.

Расчетно-пояснительная записка должна содержать постановку введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательский раздел, заключение, список литературы, приложения.

2.2 Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

На защиту проекта должна быть представлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчётные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведённых исследований.

Дата выдачи задания « ___ » _____ 20__ г.

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

Студент _____ Е.А.Панафидин
(Подпись, дата) (И.О.Фамилия)

Оглавление

Введение

Работая с операционной системой , пользователю нередко требуется посмотреть журнал запуска приложений, как в целях отслеживания нежелательных программ, так и для понимания того что было запущено вследствие каких-то определенных действий. Данная курсовая работа посвящена работе с системным вызовом exec и его перехвату в целях создания журнала запускаемых приложений.

Целью данной работы является разработка загружаемого модуля ядра для мониторинга запуска приложений путем перехвата системного вызова exec.

1. Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу, необходимо разработать загружаемый модуль ядра для мониторинга системного вызова exec.

Для решения поставленной задачи необходимо:

- провести анализ методов перехвата системных вызовов
- разработать алгоритм перехвата системного вызова exec и структуру ПО
- реализовать ПО
- провести исследование ПО

1.2 Системные вызовы

Системные вызовы – это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы. Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями. В этом режиме работает

код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи.

Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы. В большинстве операционных систем системный вызов осуществляется командой программного прерывания. Программное прерывание – это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.

Запуск на выполнение любой программы в Linux осуществляется любой из функций семейства системных вызовов exec():

- execl();
- execlp();
- execle();
- execv();
- execvp();
- execvpe().

Все вышеперечисленные вызовы являются «оберткой» для execve().

1.3 Анализ методов перехвата системных вызовов

1.3.1 Tracepoints

Точки трассировки — это датчики, статически включённые в определённые места ядра во время его компиляции[4]. Каждый такой датчик можно включить независимо от других, в результате чего он будет выдавать уведомления в тех случаях, когда достигается то место кода ядра, в которое он внедрён. Ядро содержит несколько подходящих нам точек трассировки, код которых выполняется в различные моменты работы системного вызова execve. Это
— sched_process_exec, open_exec, sys_enter_execve, sys_exit_execve.

Преимущества:

- Техническая простота реализации

Недостатки:

- Нет подробной документации
- Не заработает, если включен CONFIG_MODULE_SIG и нет закрытого ключ для подписи, находящийся у вендора дистрибутива.

1.2.2 LSM

LSM (Linux Security Modules) — это фреймворк для разработки модулей безопасности ядра. Он был создан для того, чтобы расширить стандартную модель безопасности DAC, сделать её более гибкой. Этот фреймворк использует известный модуль безопасности SELinux, а также ещё несколько других, встроенных в ядро. LSM не являются загружаемыми модулями Linux. Однако также, как и SELinux, он непосредственно интегрирован в ядро. Любое изменение исходного кода LSM требует новой компиляции ядра. Перехват в LSM реализован через набор заранее предустановленных в ядро «хуков» — функций, перехватывающих системный вызовы. LSM позволяет вставлять в код своих хуков вызов пользовательских, что позволяет безопасно работать с системными вызовами без изменения таблицы символов.

Преимущества:

- Документация

Недостатки:

- Требуется пересборка ядра, не встариваются динамически

1.2.3 kprobes

Kprobes, в отличие от kernel tracepoints, является механизмом динамического инструментирования кода.

Датчики kprobe позволяют извлекать отладочную информацию практически из любого места ядра. Они подобны особым точкам останова в коде ядра, которые выдают информацию, но при этом не останавливают выполнение кода. Датчик kprobe, в отличие от точек трассировки, можно подключить к самым разным функциям.

Принцип работы kprobes заключается в замене некоторой перехватываемой инструкции на CPU trap(int 3 для x86). При вызове данной функции генерируется исключение, по которому сохраняются регистры, а управление в свою очередь переходит к обработчику исключительной ситуации, которым и является механизм kprobes, впоследствии вызывающим обработчик, определнный программистом[3].

Выделяют 3 разновидности kprobes:

- kprobes —позволяет прервать любое место ядра, при этом для получения аргументов функции или значения каких-то переменных потрубуется вручную их извлекать из регистров.
- jprobes — jump probe, вставляется только в начало функции, но зато даёт удобный механизм доступа к аргументам прерываемой функции для нашего обработчика. Также работает не за счёт ловушек, а через setjmp/longjmp, то есть более легковесна.
- kretprobes — return probe, вставляется перед выходом из функции и даёт удобный доступ к результату функции. Реализуется через подмену адреса возврата на стеке, при этом хранение оригинального адреса происходит в отдельном буфере фиксированного размера, поэтому при частом вызове перехватываемой функции существуют риски возникновения переполнения этого буфера и отмену срабатывания kretprobes.

Преимущества:

- Документация
- Перехват практически любого места в ядре

- Оптимизированная работа механизма, скорость работы

Недостатки:

- jprobes устарел и был удален из современных версий ядер, поэтому аргументы перехватываемой функции необходимо извлекать из регистров.
- За счет специфики работы kretprobes ввиду подмены адреса возрата на стеке, возможена отмена срабатывания механизма перехвата.

1.2.4 ftrace

Название ftrace представляет собой сокращение от Function Trace — трассировка функций. Однако возможности этого инструмента гораздо шире: с его помощью можно отслеживать контекстные переключения, измерять время обработки прерываний, высчитывать время на активизацию заданий с высоким приоритетом и многое другое.
Это фреймворк, предоставляющий отладочный кольцевой буфер для записи данных. Собирают эти данные встроенные в ядро программы-трассировщики. Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию. Ftrace реализован на основе ключей компилятора -pg и -mfentry, вставляющих в начало каждой функции специальные трассировочные функции, при этом существуют оптимизация, позволяющая оставлять эти трассировочные функции только в нужных местах, а в остальных заменять их на ничего не делающие, тем самым повышая скорость работы.

Преимущества:

- скорость работы
- перехват любой функции по имени

Недостатки:

- ряд необходимых требований к конфигурации ядра, в том числе наличие самого фреймворка ftrace и списка символов kallsyms, используемого для поиска функций по имени

1.2.5 Изменение таблицы системных вызовов

Прямой доступ к адресному пространству ядра обеспечивает файл устройства /dev/kmem. В этом файле отображено все доступное виртуальное адресное пространство, включая раздел подкачки (swap-область). Открыв стандартным способом /dev/kmem, появляется возможность обратиться к любому адресу в системе, задав его как смещение в этом файле. Обращение к системным функциям осуществляется посредством загрузки параметров функции в регистры процессора и последующим вызовом программного прерывания 0x80. Обработчик этого прерывания, функция system_call, помещает параметры вызова в стек, извлекает из таблицы sys_call_table адрес вызываемой системной функции и передает управление по этому адресу. Имея полный доступ к адресному пространству ядра, можно получить все содержимое таблицы системных вызовов, т.е. адреса всех системных функций. Сохранив адрес старого обработчика и заменив его в таблице системных вызовов на новую функцию, можно добиться перехвата любого системного вызова.
Преимущества:

- Полный контроль над любыми системными вызовами
- Не зависит от конфигурации ядра

Недостатки:

- Техническая сложность реализации

- Не все обработчики можно перехватить в старых версия ядра, вследствие ряда применяемых там оптимизаций.

1.2.6 Сравнение методов перехвата системных вызовов

Выбор между методами будет производиться по следующим критериям:

- Наличие подробной документации
- Динамическая загрузка модуля ядра
- Техническая простота реализации

Результат сравнения представлен в таблице 1.

Метод	Документация	Динамическая загрузка	Техническая простота реализации
Ftrace	+	+	+
Kprobes	+	+	+
LSM	+	-	+
Kernel tracepoints	-	+	+
Модификация таблицы sys_call_table	-	+	-

Таблица 1. Сравнение методов перехвата системного вызова exec

Выводы

На основе проведенного сравнительного анализа в соответствии с выбранными критериями для реализации поставленной задачи следует использовать фреймворк ftrace, т.к он обладает подробной документацией и всем необходимым функционалом.

2. Конструкторский раздел

2.1 IDEF0

На рисунке 1 представлена диаграмма IDEF0.

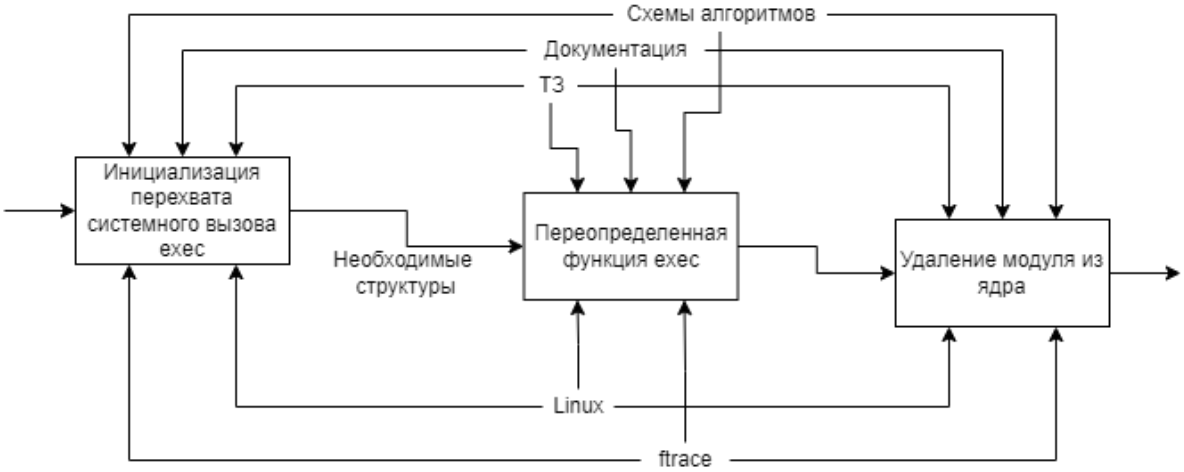

Рисунок 1. Диаграмма IDEF0

2.2 Схема алгоритма инициализации перехвата системного вызова exec

На рисунке 2 представлен алгоритм инициалзации перехвата системного вызова exec.

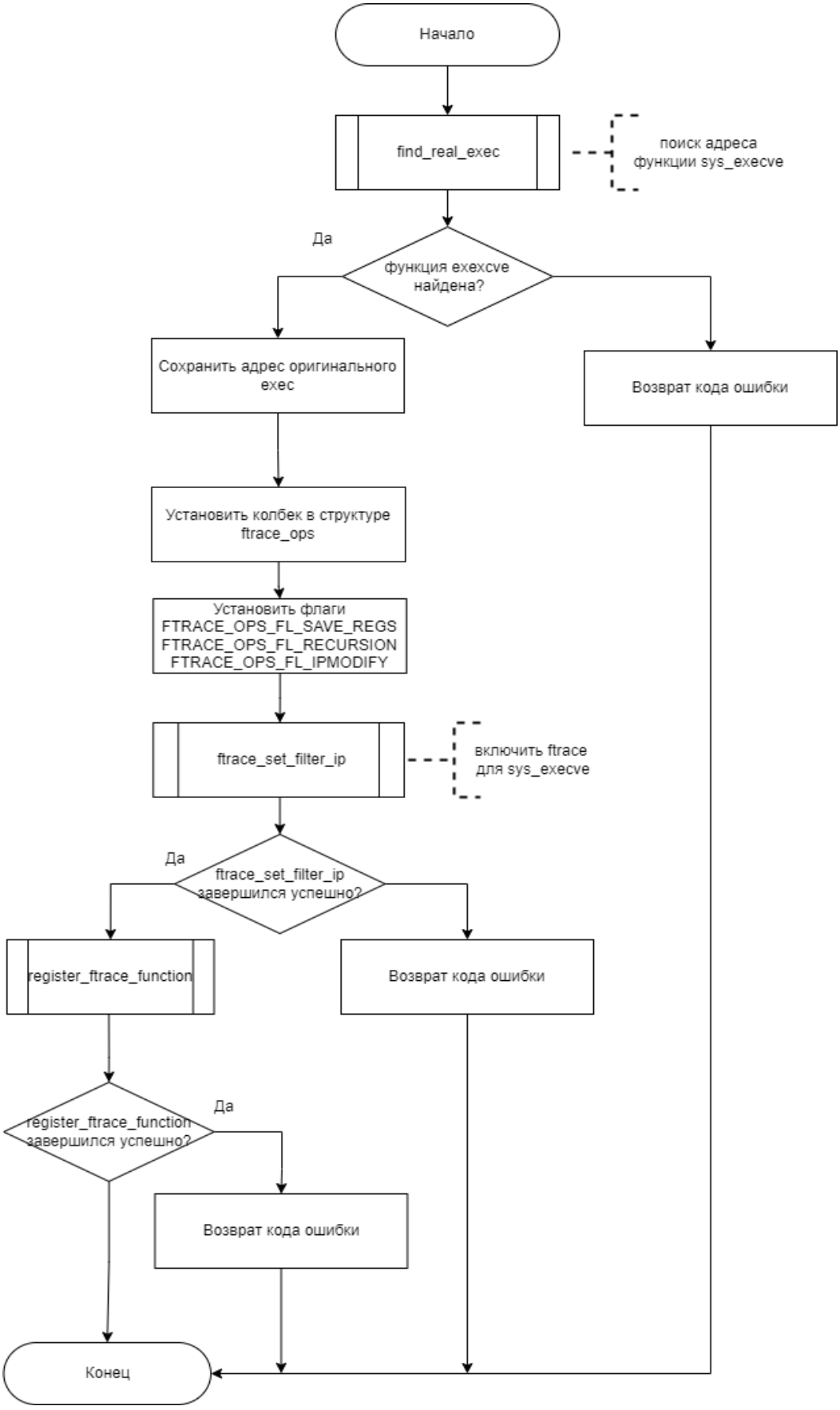

Рисунок 2. Алгоритм инициализации перехвата системного вызова exec

2.3 Схема алгоритма получения адреса системного вызова exec

На рисунке 3 представлен алгоритм получения адреса системного вызова exec.

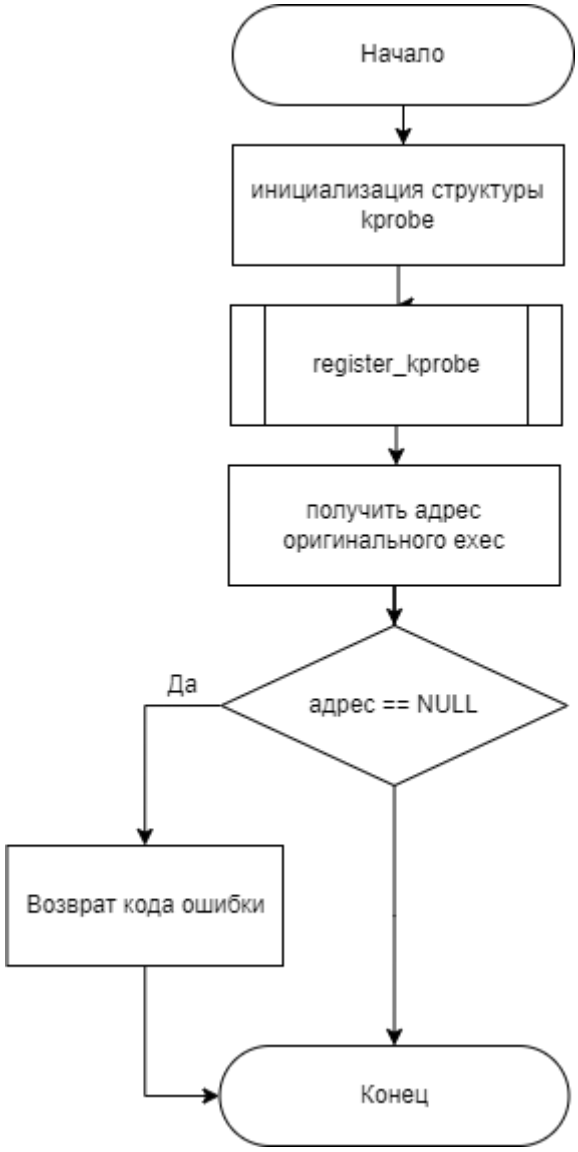

Рисунок 3. Алгоритм получения адреса системного вызова exec

2.4 Схема алгоритма обновления количества предков, вызвавших exec

На рисунке 4 изображен алгоритм обновления количества предков, вызвавших exec.

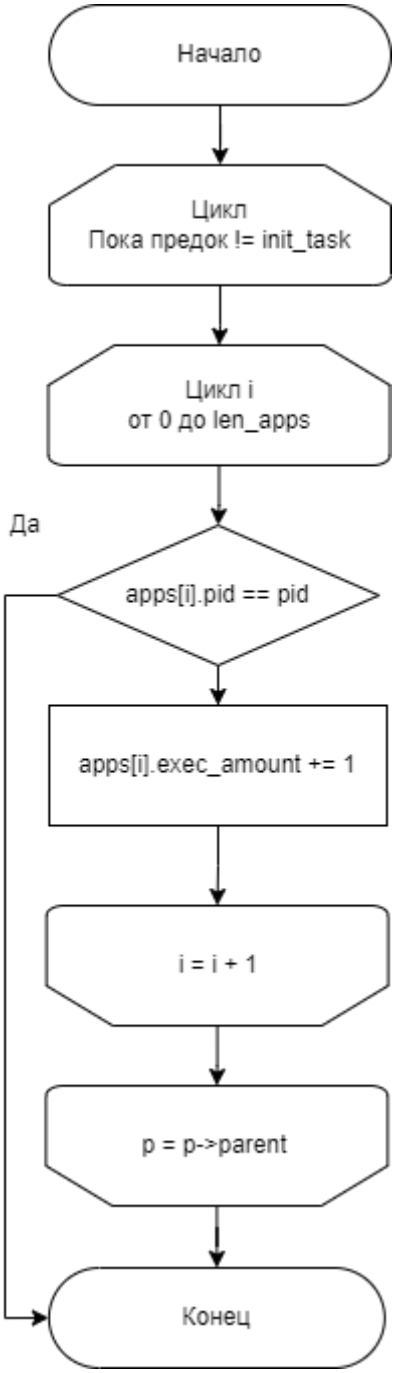

Рисунок 4. Схема алгоритма обновления количества предков, вызвавших exec.

2.5 Схема алгоритма работы переопределенного exec

На рисунке 5 изображена схема алгоритма работы переопределенного exec.

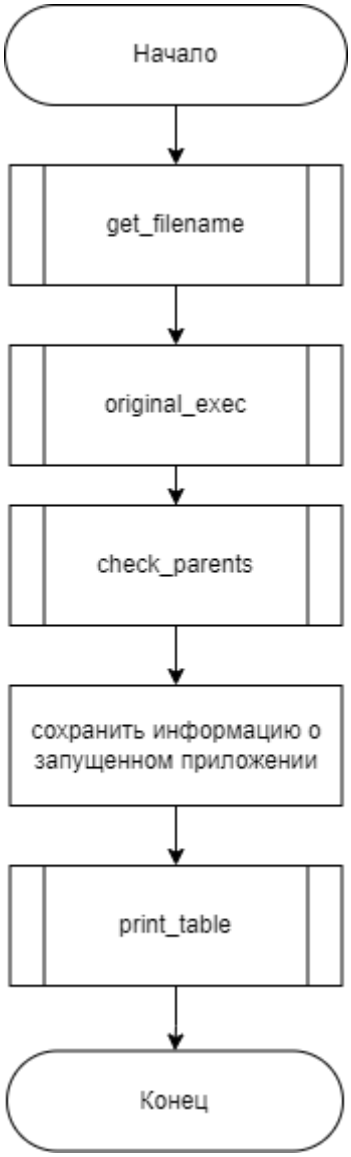

Рисунок 5. Схема алгоритма работы переопределенного exec.

2.6 Схема алгоритма удаления хуков ftrace

На рисунке 6 изображения схема алгоритма удаления хуков ftrace.

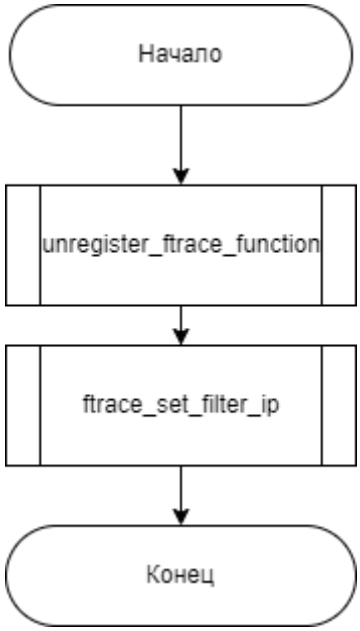

Рисунок 6. Схема алгоритма удаленния хуков ftrace.

2.2 Выводы

В результате была разработана следующая структура ПО, представленная на рисунке 7.

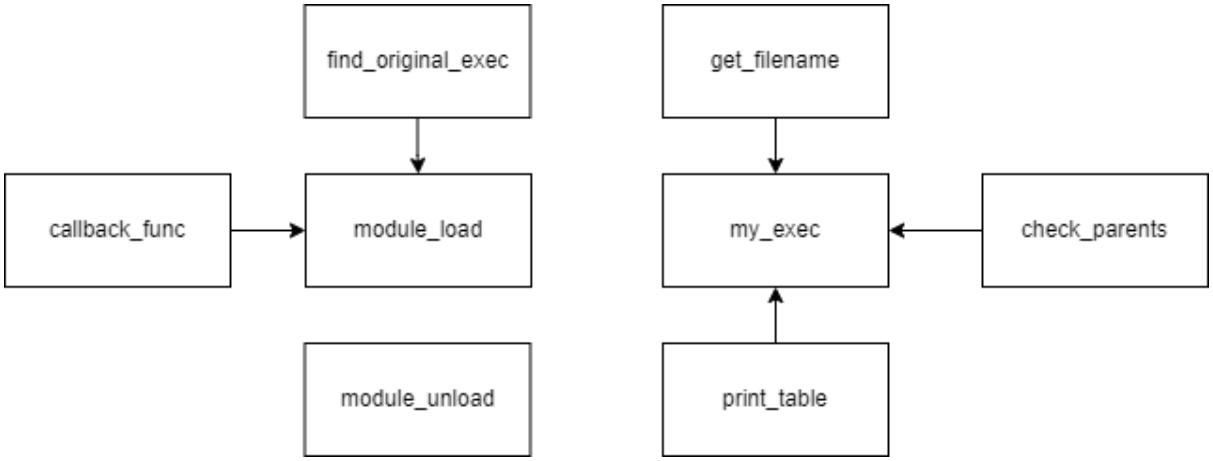

Рисунок 7. Структура ПО

3. Технологический раздел

3.1 Выбор языка и среды программирования

Для написания ПО был выбран язык C, потому что весь исходный код системы Linux написан на данном языке.
В качестве среды разработки была выбрана IDE CLion от JetBrains, она обладает всеми вспомогательными функциями для комфортной разработки: подсветкой синтаксиса, автодополнением и удобной навигацией по функциям.

3.2 Описание структур

На листинге 1 представлена структура ftrace_ops[2].

```
struct ftrace_ops ops = {
      .func                    // функция-колбек
      .flags                   // флаги
      .private                 // приватная структура
};
```

Листинг 1. Структура ftrace_ops

На листинге 2 представлена структура application.

```
struct application{
      char* name; // имя файла
      int ppid;   // идентификатор процесса-предка
      int pid;    // идентификатор процесса
      int exec_amount; // количество потомков,
вызвавших exec
}
```

Листинг 2. Структура application

3.3 Реализация функции инициализации перехвата системного вызова exec

На листинге 3 представлена реализация функции инициализации перехвата системного вызова exec, в соответствии с рисунком 1.

```
static int module_load(void)
{

	orig_exec = &original_exec;

	my_exec = my_exec_func;

	if (find_original_exec())
		return ERR;

	ops.func = callback_func;

	ops.flags = FTRACE_OPS_FL_SAVE_REGS
			| FTRACE_OPS_FL_RECURSION
			| FTRACE_OPS_FL_IPMODIFY;

	if (ftrace_set_filter_ip(&ops, address, 0, 0)) {

		return ERR;
	}

	if (register_ftrace_function(&ops)) {

		ftrace_set_filter_ip(&ops, address, 1, 0);
		return ERR;
	}
	pr_info("LKM loaded\n");

	return 0;
}
```

Листинг 3. Функция инициализация перехвата системного вызова exec

3.4 Реализация функции получения адреса оригинального exec

На листинге 4 представлена реализация функции получения адреса оригинального exec в соответствии с рисунком 2.

```
static int find_original_exec(void)
{
	struct kprobe kp = {
		.symbol_name = "__x64_sys_execve"
```

```
	};

	register_kprobe(&kp);

	address = kp.addr;
	unregister_kprobe(&kp);
	if (!address) {

		return -ENOENT;
	}

	*((unsigned long*)orig_exec) = address +
MCOUNT_INSN_SIZE;

	return 0;
}
```

Листинг 4. Функция получения адреса оригинального exec

3.5 Реализация функции обновления количества предков, вызвавших exec

На листинге 5 представлена реализация функции получения адреса оригинального exec в соответствии с рисунком 3.

```
void check_parents(struct task_struct *p) {
	int pid;
	int i;
	while (p->parent != &init_task) {
		pid = p->parent->pid;

		for(i = 0; i < len_apps; i++) {
			if (apps[i].pid == pid) {
				apps[i].exec_amount += 1;
				break;
			}

		}
		p = p->parent;
	}
```

```
}
```

Листинг 5. Функция обновления количества предков, вызвавших exec

3.6 Реализация функции переопределенного exec

На листинге 6 представлена реализация функции получения адреса оригинального exec в соответствии с рисунком 4.

```
static asmlinkage long my_exec_func(struct pt_regs
*regs)
{
	long ret;
	char *kernel_filename;

	kernel_filename = get_filename((void*) regs->di);

	char kmsg[STRING_SIZE];

	ret = original_exec(regs);

	check_parents(current);

	apps[len_apps].name = kernel_filename;
	apps[len_apps].ppid = current->parent->pid;
	apps[len_apps].exec_amount = 0;
	apps[len_apps].pid = current->pid;

	len_apps++;

	print_table();

	return ret;
}
```

Листинг 6. Функция переопределенного exec

3.7 Реализация функции удаления хуков ftrace

На листинге 7 представлена реализация функции получения адреса оригинального exec в соответствии с рисунком 5.

```
static void module_unload(void)
{

	unregister_ftrace_function(&ops);

	ftrace_set_filter_ip(&ops, address, 1, 0);

	pr_info("LKM unloaded\n");

}
```

Листинг 7. Функция удаления хуков ftrace

3.8 Makefile

На листинге 8 представлен makefile для компиляции загружаемого модуля ядра.

```
KERNEL_PATH ?= /lib/modules/$(shell uname -r)/build

obj-m += os.o

ftrace_hook:
	make -C $(KERNEL_PATH) M=$(PWD) modules

clean:
	make -C $(KERNEL_PATH) M=$(PWD) clean
```

Листинг 8. Makefile

3.9 Пример работы программы

На рисунке 8 представлен пример работы программы.

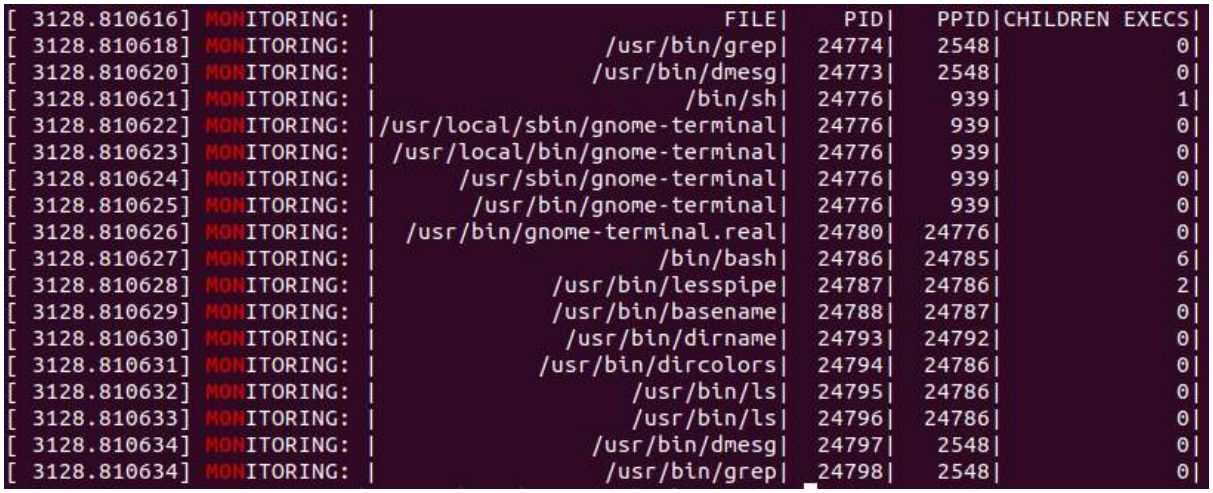

Рисунок 8. Пример работы программы

3.10 Выводы

В данном разделе было представлено обоснование выбора языка C и среды разработки CLion, а также продемонстрированы листинги основных функций и результат работы программы.

4. Исследовательский раздел

4.1 Запуск терминала

После инициализации загружаемого модуля был открыт новый терминал.

Результат представлен на рисунке 9.

Рисунок 9. Пример работы программы

Можно заметить, что для приложения /bin/bash было вызвано 4 exec среди потомков, все они представлены на скриншоте.

4.2 Запуск программы

Для дальнейшего исследования работы разработанного ПО была написано следующая программа, представленная на листинге 9.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

#include <stdlib.h>
int main() {
    int err;
    printf("Before exec\n");
    execl("ls", "1", "", "", NULL);
    printf("after exec\n");

    fork();

    printf("Before exec2\n");
    execl("ls", "2", "", "", NULL);
    printf("after exec2\n");

    return 0;
}
```

Листинг 9. Программа для исследования.

После запуска работы данной программы в уже ранее открытом терминале получаем следующий результат, представленный на рисунке 9.

```
avare@avare-VirtualBox:/media/sf_shared$ ./a.out
Before exec
after exec
Before exec2
after exec2
avare@avare-VirtualBox:/media/sf_shared$ Before exec2
after exec2
```

Рисунок 10. Результат работы.

При этом логи мониторинга запускаемых приложений выглядят следующим образом, представленным на рисунке 11.

```
[ 2610.348101] MONITORING: |                          FILE|    PID|   PPID|CHILDREN EXECS|
[ 2610.348103] MONITORING: |                     /bin/bash|   6210|   6209|             7|
[ 2610.348105] MONITORING: |              /usr/bin/lesspipe|   6211|   6210|             2|
[ 2610.348106] MONITORING: |              /usr/bin/basename|   6212|   6211|             0|
[ 2610.348107] MONITORING: |               /usr/bin/dirname|   6214|   6213|             0|
[ 2610.348108] MONITORING: |             /usr/bin/dircolors|   6215|   6210|             0|
[ 2610.348109] MONITORING: |                 /usr/bin/dmesg|   6216|   3164|             0|
[ 2610.348110] MONITORING: |                  /usr/bin/grep|   6217|   3164|             0|
[ 2610.348111] MONITORING: |                        ./a.out|   6220|   6210|             0|
[ 2610.348112] MONITORING: |                             ls|   6220|   6210|             0|
[ 2610.348113] MONITORING: |                             ls|   6220|   6210|             0|
[ 2610.348113] MONITORING: |                             ls|   6221|    885|             0|
```

Рисунок 11. Системные логи

Таким образом, во-первых, у приложения /bin/bash увеличилось количество exec, вызванного среди потомков, на 3, среди них запуск программы a.out и внутри нее два вызова exec команды ls.
Во-вторых, после вызова fork() внутри программы мы образовали дополнительный поток, в котором выполнился третий exec, при этом у него можно заметить иной pid и ppid.

4.3 Выводы

В данном разделе было проведено исследование работы разработанного ПО при обычном запуске приложений, а также в условиях нескольких потоков. Исследование подтвердило правильность работы реализованной программы.

Заключение

В процессе выполнения курсового проекта по операционным системам для реализации поставленной задачи был проведен сравнительный анализ методов перехвата системного вызова exec и выбран фреймворк ftrace, разработаны необходимые алгоритмы и структура ПО, реализована программа загружаемого модуля ядра для мониторинга запускаемых приложений, а также проведенно исследование разработанного ПО в условиях многопоточности.

Список литературы

1. Документация ftrace [Электронный ресурс]. – Режим доступа: URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (дата обращения: 19.12.2021)
2. Механизмы профилирования Linux [Электронный ресурс]. – Режим доступа: URL: https://habr.com/ru/company/metrotek/blog/261003/(дата обращения: 19.12.2021)
3. О сложностях мониторинга работающих процессов в Linux [Электронный ресурс]. – Режим доступа: URL: https://bookflow.ru/o-slozhnostyah-monitoringa-rabotayushhih-protsessov-v-linux/ (дата обращения: 19.12.2021)

Приложение

```
#define STRING_SIZE 1000
#include <linux/kprobes.h>
#include <linux/ftrace.h>
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/types.h>

#define ERR 1
MODULE_DESCRIPTION("exec");
MODULE_AUTHOR("Egor Panafidin");
MODULE_LICENSE("GPL");


typedef struct {
	char* name;
	int ppid;
	int pid;
	int exec_amount;
} application;


application apps[10000];
int len_apps = 0;


void *orig_exec;
void *my_exec;
unsigned long address;
struct ftrace_ops ops;



static char *get_filename(const char __user *filename)
{
	char *name;

	name = kmalloc(50, GFP_KERNEL);

	if (strncpy_from_user(name, filename, 50) < 0) {
		kfree(name);
		return NULL;
	}

	return name;
}

static asmlinkage long (*original_exec)(struct pt_regs *regs);

void check_processes(int pid) {
```

```
	int i;
	for(i = 0; i < len_apps; i++) {
		if (apps[i].pid == pid) {
			apps[i].exec_amount += 1;
			break;
		}

	}
}
void check_parents(struct task_struct *p) {
	int pid;
	int i;
	while (p->parent != &init_task) {
		pid = p->parent->pid;

		for(i = 0; i < len_apps; i++) {
			if (apps[i].pid == pid) {
				apps[i].exec_amount += 1;
				break;
			}

		}
		p = p->parent;
	}

}


static int find_original_exec(void)
{
	struct kprobe kp = {
		.symbol_name = "__x64_sys_execve"
	};

	register_kprobe(&kp);

	address = kp.addr;
	unregister_kprobe(&kp);
	if (!address) {

		return -ENOENT;
	}

	*((unsigned long*)orig_exec) = address + MCOUNT_INSN_SIZE;

	return 0;
}

static void notrace callback_func(unsigned long ip, unsigned long
parent_ip,
		struct ftrace_ops *ops, struct ftrace_regs *fregs)
{
	struct pt_regs *regs = ftrace_get_regs(fregs);

	if (!within_module(parent_ip, THIS_MODULE))
```

```
                regs->ip = (unsigned long)my_exec;
}

void print_table(void) {
        pr_info("-------------------------------------------");
        pr_info("|%30s|%7s|%7s|%14s|\n", "FILE", "PID","PPID",
"CHILDREN EXECS");
        int i;
        for(i = 0; i < len_apps; i++) {
                pr_info("|%30.30s|%7d|%7d|%14d|\n", apps[i].name,
apps[i].pid, apps[i].ppid, apps[i].exec_amount);
        }
}


static asmlinkage long my_exec_func(struct pt_regs *regs)
{
        long ret;
        char *kernel_filename;

        kernel_filename = get_filename((void*) regs->di);

        char kmsg[STRING_SIZE];

        ret = original_exec(regs);

        check_parents(current);

        apps[len_apps].name = kernel_filename;
        apps[len_apps].ppid = current->parent->pid;
        apps[len_apps].exec_amount = 0;
        apps[len_apps].pid = current->pid;

        len_apps++;

        print_table();

        return ret;
}


static int module_load(void)
{

        orig_exec = &original_exec;

        my_exec = my_exec_func;

        if (find_original_exec())
                return ERR;

        ops.func = callback_func;

        ops.flags = FTRACE_OPS_FL_SAVE_REGS
                        | FTRACE_OPS_FL_RECURSION
```

```
                        | FTRACE_OPS_FL_IPMODIFY;

        if (ftrace_set_filter_ip(&ops, address, 0, 0)) {

                return ERR;
        }

        if (register_ftrace_function(&ops)) {

                ftrace_set_filter_ip(&ops, address, 1, 0);
                return ERR;
        }
        pr_info("LKM loaded\n");

        return 0;
}


static void module_unload(void)
{

        unregister_ftrace_function(&ops);

        ftrace_set_filter_ip(&ops, address, 1, 0);

        pr_info("LKM unloaded\n");

}

module_init(module_load);
module_exit(module_unload);
```