Оглавление

Введение

Процесс является одним из самых важных компонентов операционной системы Linux, так как представление вычислительной задачи в Linux построено именно вокруг абстракции процесса. В связи такой высокой ролью процесса, зачастую возникает необходимость получения некоторой информации о процессах, такой как ресурсы, выделенные процессу (память, используемые файлы и др.), иерархия процессов и т.д. К сожалению, ОС Linux не предоставляет удобных средств для сбора и вывода данной информации.

По этой причине была поставлена цель – в рамках курсового проекта разработать загружаемый модуль ядра и пользовательское приложение, предоставляющие информацию о процессах и самой операционной системе Linux.

1. Аналитический раздел

В данном разделе представлены постановка задачи, анализ подходов реализации, анализ работы с модулями ядра и выбор подходящего решения.

1.1 Обзор технического задания

В соответствии с техническим заданием необходимо разработать программное обеспечение, позволяющее получить информацию о процессах и текущих параметрах системы, а именно:

- имя компьютера;
- имя пользователя;
- время непрерывной работы системы;
- модель процессора;
- частота процессора;
- загруженность процессора;
- объем всей оперативной памяти;
- объем используемой оперативной памяти.

Отдельно получать информацию о запущенных программах, а именно:

- идентификатор процесса;
- идентификатор родителя;
- используемые файлы (имя, размер);
- используемая память в куче (адрес, размер);

1.2 Анализ подходов реализации

Операционная система Linux предоставляет два подхода для получения информации о процессах:

- чтение информации из каталогов файловой системы procfs;
- получение информации в пространстве ядра.

Чтение информации из каталогов файловой системы procfs возможно реализовать в пространстве пользователя.

Второй вариант подразумевает под собой написание модуля ядра. Первый вариант позволяет получить меньшее количество информации о процессах, нежели второй. Тем более, чтение информации из каталогов файловой системы procfs может влиять на состояние процессов, что может негативно сказаться на работе всей системы. Поэтому предпочтение отдается варианту, основанному на получение информации непосредственно в пространстве ядра [1].

1.3 Получение информации о системе в режиме пользователя

Операционная система Linux предоставляет широкие возможности для получения различной системной информации, часто без необходимости при этом обладать правами суперпользователя. Существует множество встроенных утилит и команд, которые позволяют получить подробную информацию о самой операционной системе, а также об аппаратном обеспечении компьютера и т.п.

Утилита lspci предназначена для вывода информации обо всех PCI-шинах в системе, а также обо всех устройствах, присоединенных к этим шинам. По умолчанию она показывает краткий список таких устройств. Однако существует возможность использовать многочисленные опции lspci для получения более подробной информации или информации, ориентированной на последующую обработку с помощью других программ.

Команда dmesg обычно используется в Linux для того, чтобы просмотреть содержимое кольцевого буфера ядра. Она позволяет пользователю вывести содержание сообщений, выдаваемых в процессе загрузки системы. Утилита lspci хорошо помогает при обнаружении PCI-устройств, однако нам часто требуется список всех устройств в системе. Используя dmesg, мы можем просмотреть характеристики всех устройств, которые обнаружены нашей операционной системой.

Для получения информации об оперативной памяти или центральном процессоре в реальном времени на работающей системе можно также воспользоваться информацией в каталоге виртуальной файловой системы procfs [2]. Выполнив команду ls в корневом каталоге procfs (обычно /proc), можно увидеть различные директории и файлы, которые содержат информацию о системе.

Программа fdisk – это инструмент для работы с таблицей разбиения диска. Физические диски обычно разбиваются на несколько логических дисков, которые называются разделами диска. Информация о разбиении физического диска на разделы хранится в таблице разбиения диска, которая находится в нулевом секторе физического диска. Если имеется два или более дисков (например, hda и hdb), и необходимо получить данные о конкретном диске, нужно указать в команде желаемый диск, например fdisk -l /dev/hda.

Утилита dmidecode выводит содержимое таблицы DMI (Desktop Management Interface) системы в формате, предназначенном для восприятия человеком. Эта таблица содержит информацию, относящуюся к компонентам аппаратного обеспечения системы, а также сведения о версии BIOS и т.д. В выводе dmidecode не только содержится описание текущей конфигурации системы, но и приводятся данные о максимально допустимых значениях

параметров, например, о поддерживаемых частотах работы CPU, максимально возможном объеме памяти и так далее.

1.3.1 Выбор метода получения информации о системе в режиме пользователя

Вся необходимая информация о системе в удобном виде предоставляется файловой системой *procfs*. Это обосновывает выбор *procfs* как средства получения информации. Важно также отметить, что для получения этой информации не требуются права суперпользователя.

1.3.2 Получение информации из виртуальной файловой системы procfs

Для того чтобы узнать собственное имя компьютера с помощью *procfs*, независимое от сетевых интерфейсов, необходимо прочитать информацию из файла */proc/sys/kernel/hostname*

Для того чтобы узнать модель и частоту центрального процессора, необходимо проанализировать файл */proc/cpuinfo*, в котором хранится информация о процессоре и его состоянии в реальном времени. Пример такого файла представлен на рисунке 1.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
stepping        : 9
microcode       : 0xd6
cpu MHz         : 2304.000
cache size      : 4096 KB
```

Рисунок 1. Фрагмент файла */proc/cpuinfo*.

Модель процессора считывается из поля *model name*, а частота из поля *cpu MHz*.

Для того чтобы вычислить процент загруженности центрального процессора, следует проанализировать файл */proc/stat*, в котором находится информация об активности процессора. Пример такого файла представлен на рисунке 2.

```
cpu  16402 1012 10207 409800 934 0 96 0 0 0
cpu0 8364 422 5345 204757 458 0 23 0 0 0
cpu1 8038 590 4862 205043 476 0 72 0 0 0
intr 1876553 2 2538 0 0 0 0 0 0 1 4171 0 0 105407
```

Рисунок 2. Фрагмента файла */proc/stat*

Необходимая для вычисления процента загруженности информация хранится в первых четырех полях строчки *cpu*. Перечислим их значение слева направо:

- число процессов, выполняющихся в режиме пользователя;
- число процессов с изменённым приоритетом (*nice*), выполняющихся в режиме пользователя;
- число процессов, выполняющихся в режиме ядра;
- число процессов, выполняющих функцию простоя процессора (*idle*).

Для того чтобы получить информацию об оперативной памяти, следует проанализировать файл */proc/meminfo*. Пример такого файла представлен на рисунке 3.

```
MemTotal:        2035424 kB
MemFree:          242872 kB
MemAvailable:    1031100 kB
```

Рис. 3. Фрагмент файла */proc/meminfo*.

Объем всей оперативной памяти считывается из поля *MemTotal*, а объем памяти, доступной для немедленного её выделения процессам из поля

MemAvailable. Таким образом, объем используемой оперативной памяти вычисляется из разности этих значений.

1.4 Загружаемые модули ядра Linux

Ядро Linux относится к категории так называемых монолитных – это означает, что большая часть функциональности операционной системы реализована ядром и запускается в привилегированном режиме. Этот подход отличен от подхода микроядра, когда в режиме ядра выполняется только основная функциональность (взаимодействие между процессами [inter-process communication, IPC], диспетчеризация, базовый ввод-вывод [I/O], управление памятью), а остальная функциональность вытесняется за пределы привилегированной зоны (драйверы, сетевой стек, файловые системы). Ядро Linux динамически изменяемое – это означает, что можно загружать в ядро дополнительную функциональность, выгружать функции из ядра и даже добавлять новые модули, использующие другие модули ядра. Преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули.

1.4.1 Устройство модуля ядра

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (main) в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии 2.6 эти функции можно именовать как угодно). Функция входа вызывается, когда модуль загружается в ядро, а функция выхода – соответственно при выгрузке из ядра. Поскольку функции входа и выхода определяются программистом, для указания назначения этих функций используются макросы module_init и module_exit. Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание

модуля, а также другие параметры. Пример простого загружаемого модуля приведен на рисунке 4.

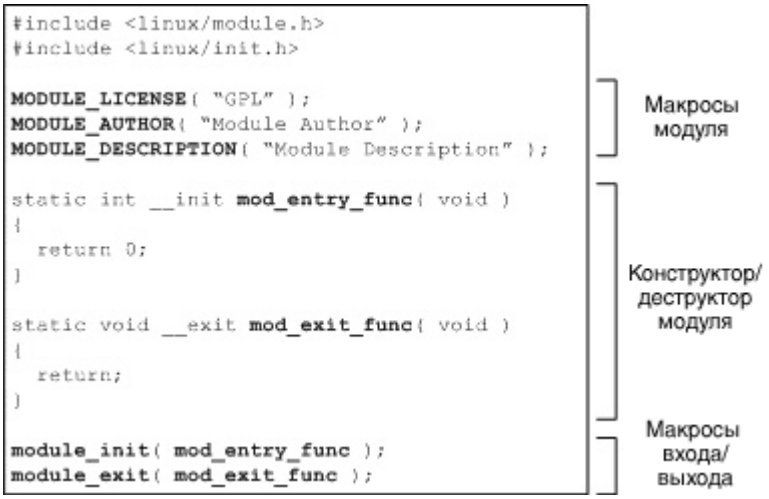

Рисунок 4. Пример загружаемого модуля с разделами ELF

1.4.2 Объектный код модуля ядра

Загружаемый модуль представляет собой специальный объектный файл в формате ELF (Executable and Linkable Format). Обычно объектные файлы обрабатываются компоновщиком, который разрешает символы и формирует исполняемый файл. Однако в связи с тем, что загружаемый модуль не может разрешить символы до загрузки в ядро, он остается ELF-объектом. Для работы с загружаемыми модулями можно использовать стандартные средства работы с объектными файлами (которые в версии 2.6 имеют суффикс .ko, от kernel object). Например, если вывести информацию о модуле утилитой objdump, можно обнаружить несколько привычных разделов, в том числе .text (инструкции), .data (инициализированные данные) и .bss (Block Started Symbol или неинициализированные данные). В модуле также обнаружатся дополнительные разделы, ответственные за поддержку его динамического поведения. Раздел .init.text содержит код module_init, а раздел .exit.text – код module_exit code (рисунок 5). Раздел .modinfo содержит тексты макросов, указывающие тип лицензии, автора, описание и т.д.

Рисунок 5. Пример структуры загружаемого модуля с разделами ELF

1.4.3 Жизненный цикл загружаемого модуля ядра

Процесс загрузки модуля начинается в пользовательском пространстве с команды insmod (вставить модуль). Команда insmod определяет модуль для загрузки и выполняет системный вызов уровня пользователя init_module для начала процесса загрузки.

Функция init_module работает на уровне системных вызовов и вызывает функцию ядра sys_init_module (рисунок 6). Это основная функция для загрузки модуля, обращающаяся к нескольким другим функциям для решения специальных задач. Аналогичным образом команда rmmod выполняет системный вызов функции delete_module, которая обращается в ядро с вызовом sys_delete_module для удаления модуля из ядра.

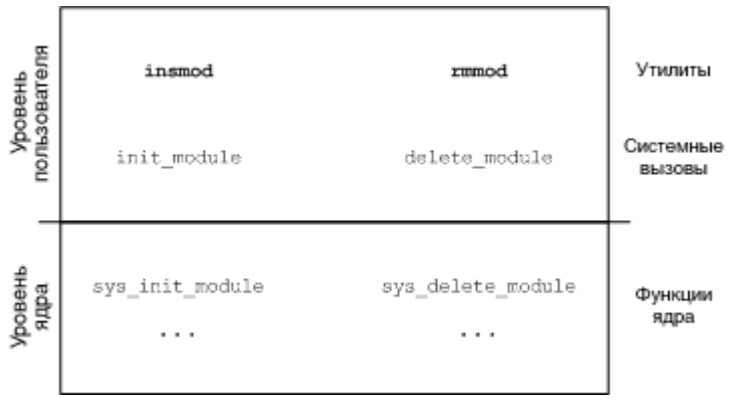

Рисунок 6. Основные команды и функции, участвующие в загрузке и выгрузке модуля.

1.4.4 Загрузка модуля

Внутренние функции для загрузки модуля представлены на рисунке 6. При вызове функции ядра sys_init_module сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения (при помощи функции capable). Затем вызывается функция load_module, которая выполняет работу по размещению модуля в ядре и производит необходимые операции. Функция load_module возвращает ссылку, которая указывает на только что загруженный модуль. Затем он вносится в двусвязный список всех модулей в системе, и все потоки, ожидающие изменения состояния модуля, уведомляются при помощи специального списка. В конце вызывается функция init() и статус модуля обновляется, чтобы указать, что он загружен и доступен. Внутренние процессы загрузки модуля представляют собой анализ и управление модулями ELF.

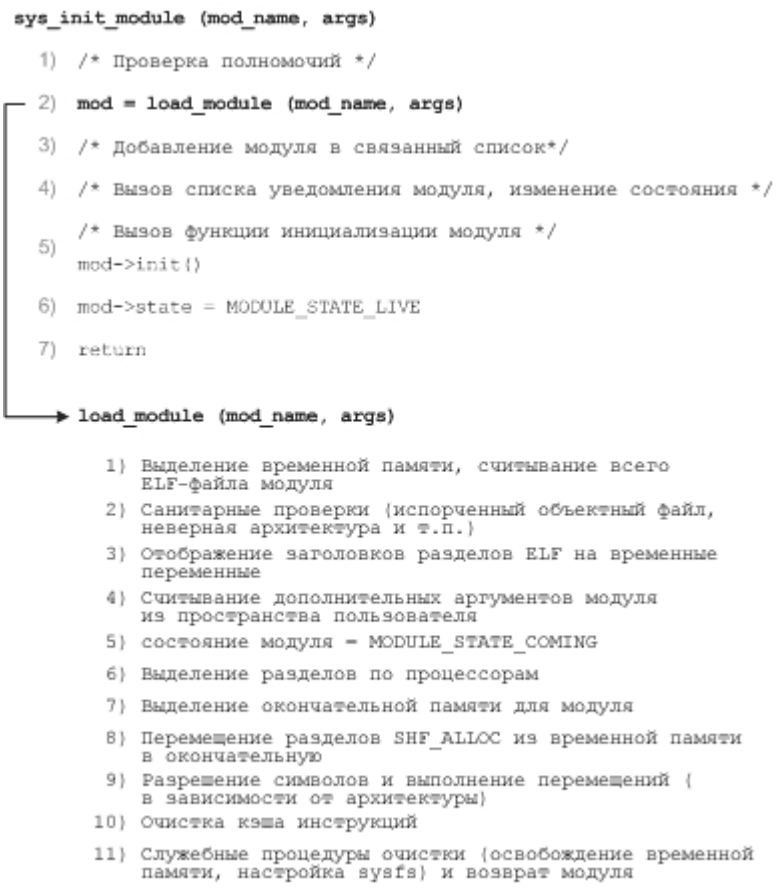

Рисунок 7. Алгоритм загрузки модуля.

Функция load_module (которая находится в ./linux/kernel/module.c) начинает с выделения блока временной памяти для хранения всего модуля ELF. Затем модуль ELF считывается из пользовательского пространства во временную память при помощи copy_from_user. Следующим шагом является ряд проверок загруженного образа (например, является ли ELF-файл допустимым? Соответствует ли он текущей архитектуре?). После того как проверка пройдена, образ ELF анализируется и создается набор вспомогательных переменных для заголовка каждого раздела, чтобы облегчить дальнейший доступ к ним. Поскольку базовый адрес объектного файла ELF равен 0 (до перемещения), эти переменные включают соответствующие смещения в блок временной памяти. Во время создания вспомогательных переменных также проверяются заголовки разделов ELF, чтобы убедиться, что загружаемый модуль корректен.

Дополнительные параметры модуля, если они есть, загружаются из пользовательского пространства в другой выделенный блок памяти ядра (шаг 4), и статус модуля обновляется, чтобы обозначить, что он загружен (MODULE_STATE_COMING). Если необходимы данные для процессоров (согласно результатам проверки заголовков разделов), для них выделяется отдельный блок. В предыдущих шагах разделы модуля загружались в память ядра (временную), и было известно, какие из них используются постоянно, а какие могут быть удалены.

На следующем шаге (7) для модуля в памяти выделяется окончательное расположение, и в него помещаются необходимые разделы (обозначенные в заголовках SHF_ALLOC или расположенные в памяти во время выполнения). Затем производится дополнительное выделение памяти размера, необходимого для требуемых разделов модуля. Производится проход по всем разделам во временном блоке ELF, и те из них, которые необходимы для выполнения, копируются в новый блок. Затем следуют некоторые служебные процедуры. Также происходит разрешение символов, как расположенных в ядре (включенных в образ ядра при компиляции), так и временных (экспортированных из других модулей). Затем производится проход по оставшимся разделам и выполняются перемещения. Этот шаг зависит от архитектуры и соответственно основывается на вспомогательных функциях, определенных для данной архитектуры (./linux/arch/<arch>/kernel/module.c). В конце очищается кэш инструкций (поскольку использовались временные разделы .text), выполняется еще несколько служебных процедур (очистка памяти временного модуля, настройка sysfs) и, в итоге, модуль возвращает load_module.

1.4.5 Выгрузка модуля

Выгрузка модуля фактически представляет собой зеркальное отражение процесса загрузки за исключением того, что для безопасного удаления модуля

необходимо выполнить несколько проверок. Выгрузка модуля начинается в пользовательском пространстве с выполнения команды rmmod (удалить модуль). Внутри команды rmmod выполняется системный вызов delete_module, который в конечном счете инициирует sys_delete_module внутри ядра (рисунок 6). Основные операции удаления модуля показаны на рисунке 8.

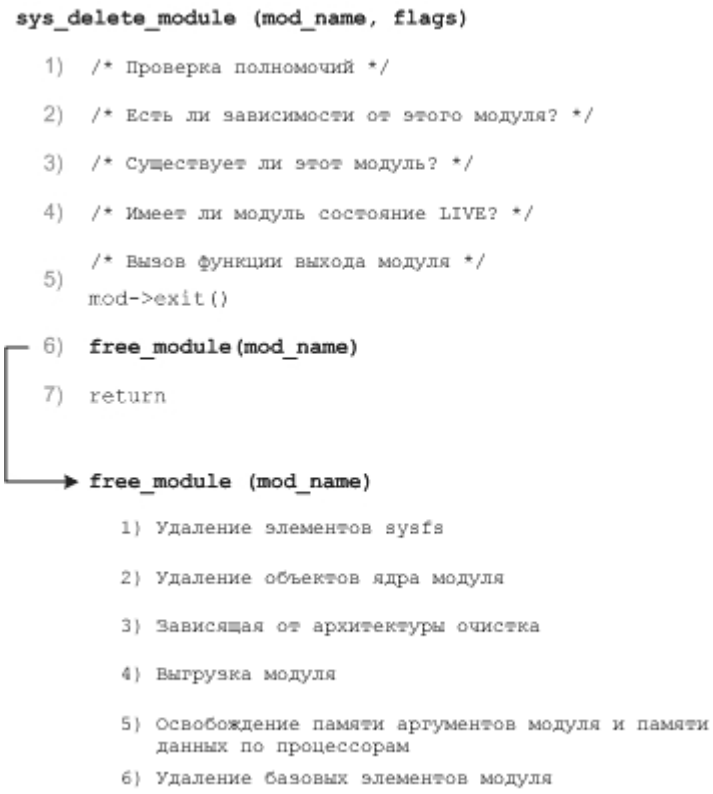

Рисунок 8. Алгоритм выгрузки модуля.

При вызове функции ядра sys_delete_module (с именем удаляемого модуля в качестве параметра) сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения. Затем по списку проверяются зависимости других модулей от данного модуля. При этом используется список modules_which_use_me, содержащий по элементу для каждого зависимого модуля. Если список пуст, т.е. зависимостей не обнаружено, то модуль становится кандидатом на удаление (иначе возвращается ошибка). Затем проверяется, загружен ли модуль. Ничто не запрещает пользователю запустить команду rmmod для модуля, который в данный момент

устанавливается, поэтому данная процедура проверяет, активен ли модуль. После нескольких дополнительных служебных проверок предпоследним шагом вызывается функция выхода данного модуля (предоставляемая самим модулем). В заключение вызывается функция free_module. К моменту вызова free_module уже известно, что модуль может быть безопасно удален. Зависимостей не обнаружено, и для данного модуля можно начать процесс очистки ядра. Этот процесс начинается с удаления модуля из различных списков, в которые он был помещен во время установки. Потом инициируется команда очистки, зависящая от архитектуры (она расположена в ./linux/arch/<arch>/kernel/module.c). Затем обрабатываются зависимые модули, и данный модуль удаляется из их списков. В конце, когда с точки зрения ядра очистка завершена, освобождаются различные области памяти, выделенные для модуля, в том числе память для параметров, память для данных по процессорам и память модуля ELF [3].

1.5 Процессы ОС Linux

Процесс – это программа в стадии выполнения. Он состоит из исполняемого программного кода, набора ресурсов (таких, как открытые файлы), внутренних данных ядра, адресного пространства, одного или нескольких потоков исполнения и секции данных, содержащей глобальные переменные. С каждым процессом связан (ассоциирован) "описатель процесса" или дескриптор процесса. Дескриптор содержит информацию, используемую для того, чтобы отслеживать процесс в оперативной памяти. В частности, в дескрипторе содержатся идентификатор процесса (PID), его состояние, ссылки на родительский и дочерние процессы, регистры процессора, список открытых файлов и информация об адресном пространстве.

Ядро Linux использует циклически замкнутый двусвязный список записей struct task_struct (Приложение А) для хранения дескрипторов процессов. Эта структура объявлена в файле linux/sched.h.

1.5.1 Представление процессов в памяти ОС Linux

Строка struct list_head tasks внутри определения struct task_struct показывает, что ядро использует циклический связанный список для хранения задач. Это означает, что можно использовать стандартные макросы и функции для работы со связанными списками с целью просмотра полного списка задач[4].

Как известно, "родителем всех процессов" в системе Linux является процесс init. Так что он должен стоять в начале списка, хотя, строго говоря, начала не существует раз речь идет о циклическом списке. Дескриптор процесса init задается статично (is statically allocated):

extern struct task_struct init_task.

Рисунок 9 иллюстрирует представление процессов в памяти в виде связанного списка.

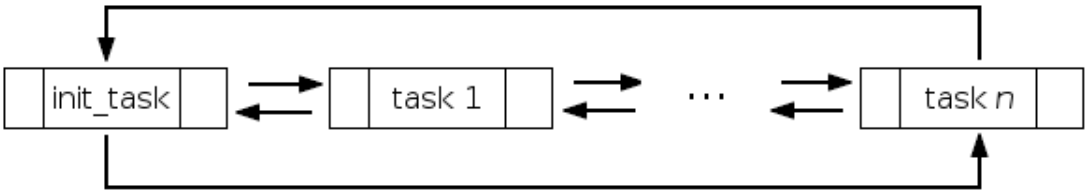

Рисунок 9. Представление процессов в виде связного списка

Имеется несколько макросов и функций, которые помогают перемещаться по этому списку:

- for_each_process() - это макрос, который проходит весь список задач;
- next_task() - макрос, определенный в linux/sched.h, возвращает следующую задачу из списка;

Определения данных макросов представлены в приложении Б[4].

1.5.2 Построение дерева процессов

Построение дерева процессов основано на иерархии “предок- потомок”. Таким образом, для корректного построения дерева необходимо рекурсивно пройти по всем потомкам процесса, начиная с init. При этом необходимо запоминать глубину рекурсии на каждом шаге для корректного отображения дерева.

1.5.3 Получение информации о памяти процесса

Адресное пространство процесса представлено полем mm типа struct mm_struct. В нашем случае, особый интерес представляет поле mmap данной структуры, которое имеет тип struct vm_area_struct. Данное поле представляет собой список областей памяти процесса. Данная структура определена в файле <linux/mm_types.h>, некоторые ее поля представлены в приложении В.

Особый интерес представляет поле vm_next, которое указывает на следующую область памяти, а также поля vm_start и vm_end, с помощью которых можно получить информацию об адресах начала и конца области памяти, а также вычислить ее размер [6].

1.5.4 Получение информации об открытых процессом файлах

Для получения информации о файлах, открытых процессом, используется структура struct fdtable, определенная в файле <linux/fdtable.h> (приложение Г). Данная структура имеет поле fd, которое определяет массива указателей на дескрипторы файлов, открытых процессом.

Таким образом, обратившись к полям каждого элемента массива fd, можно получить необходимую информацию: размер файла, путь, имя [5].

1.6 Взаимодействие модуля ядра с пространством пользователя

Для приема управляющих данных и передачи результатов функций в пользовательское приложение целесообразным будет создать три файла в каталоге /proc, каждый из которых будет являться интерфейсом

соответствующей функции разрабатываемого модуля. Интерфейс для взаимодействия с пространством пользователя представляется с помощью struct proc_dir_entry. Некоторые поля данной структуры представлены в приложении Д.

Основную работу по созданию и уничтожению имени в /proc выполняет пара вызовов, определенных в файле <linux/proc_fs.h>.

Стоит обратить внимание на поле proc_fops, типа struct file_operations. Данная структура имеет достаточно большое количество полей, однако, в нашем случае интерес представляют два из них: write и read. Данные поля определяют те функции, которые будет вызываться при попытке чтения и записи в ранее созданные имена в каталоге /proc. Прототипы данных функций представлены в листинге 1.

```
ssize_t write_proc( struct file *filp, const char *buf, size_t count,loff_t *offp);

ssize_t read_proc( struct file *filp, char *buf, size_t count,loff_t *offp );
```

Листинг 1. Прототипы функций чтения/записи

Данные функции принимают в качестве параметров указатель на файл, к которому осуществляется доступ, указатель на буфер в пространстве пользователя, размер считываемых/записываемых данных и смещение в файле. Для записи данных в буфер пользовательского пространства рекомендуется использовать функцию copy_to_user(), для чтения - copy_from_user. Данные функции определены в <linux/uaccess.h>.

Таким образом, для того чтобы обеспечить взаимодействие разрабатываемого приложения и модуля ядра, необходимо:

1. Создать функции для чтения/записи в файлы каталога /proc.
2. Для каждого из файлов определить структуры struct file_operations.

3. С помощью вызова proc_create() создать имена в /proc, передав в качестве параметра функции структуру struct file_operations, указавая тем самым, какие функции вызывать при взаимодействии приложения с созданными файлами [2].

Вывод

В данном разделе были рассмотрены постановка задачи, анализ подходов реализации, анализ работы с модулями ядра и выбор подходящего решения.

2. Конструкторский раздел

В данном разделе представлено проектирование ПО: архитектура программы, информация о функциях загружаемого модуля ядра и информация о пользовательском приложении.

2.1 Общая архитектура программы

В состав программного обеспечения входят загружаемый модуль ядра и пользовательское приложение. Для взаимодействия приложения и ядра используется файловая система procfs. Общая архитектура приложения представлена на рисунке 10.

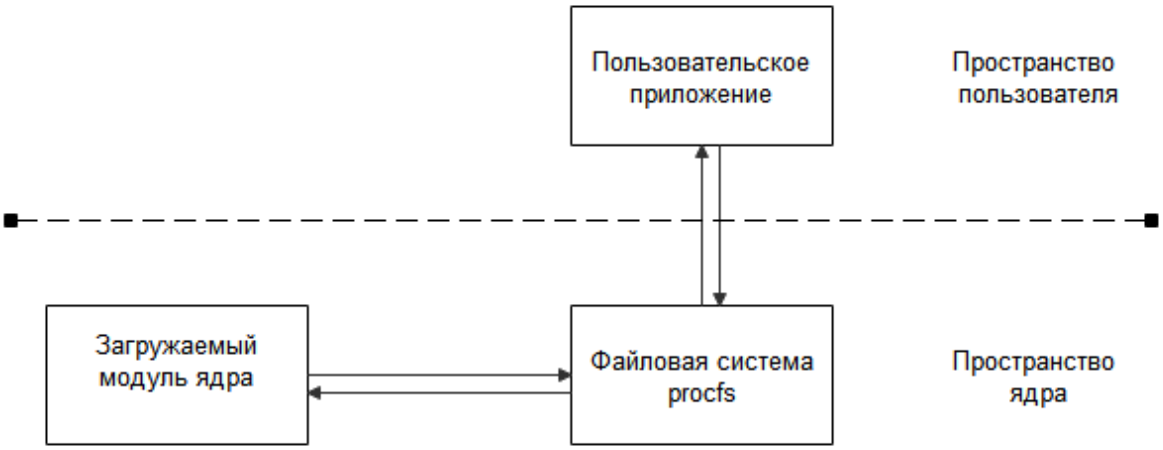

Рисунок 10. Общая архитектура приложения

2.2 Загружаемый модуль ядра

Рассмотрим алгоритм работы функций загрузки и выгрузки разрабатываемого модуля.

Разрабатываемый загружаемый модуль ядра выполняет три основных функции:

1. Построение дерева процессов;
2. Сбор информации о файлах, используемых процессом;
3. Сбор информации об адресном пространстве процесса.

Таким образом, в функции загрузки модуля необходимо определить три структуры типа struct file_operations, каждая из которых будет определять функции для чтения/записи в соответствующий файл в /proc. Затем необходимо с помощью функции proc_create создать три имени в каталоге /proc, каждое из которых будет определять один из созданных файлов. В качестве одного из аргументов функции proc_create необходимо передать структуру file_operations, чтобы определить, какие функции будут вызваны при чтении/записи в один из файлов из пространства пользователя.

Соответственно, при выгрузке модуля необходимо с помощью вызова remove_proc_entry освободить память, выделенную под структуры, определяющие имена в /proc. Код функций загрузки и выгрузки модуля представлен в приложении Е.

2.3 Пользовательское приложение

Разрабатываемое приложение предоставляет пользователю информацию о системе, обновляющуюся каждую секунду, и функции для получения информации, формируемой загружаемым модулем ядра. Процесс получения информации из пространства ядра состоит из двух этапов:

- передача управляющей информации из пространства пользователя в пространство ядра;
- чтение из пространства ядра информации, полученной в результате работы функций модуля ядра.

Для того чтобы упростить разработку пользовательского приложения, необходимо разработать единый способ передачи данных в модуль из приложения, что позволит вызывать одну и ту же функцию получения информации для получения разной информации за счет указания в качестве параметров этой функции имени файла в каталоге /proc и переменной, содержащей управляющую информацию. В качестве управляющей

информации для функций получения информации о памяти и файлах следует использовать ID процесса.

Вывод

В данном разделе было представлено проектирование ПО: архитектура программы, информация о функциях загружаемого модуля ядра и информация о пользовательском приложении.

3. Технологический раздел

В данном разделе рассмотрен выбор средств реализации, а также представлен интерфейс.

3.1 Средства реализации

Для реализации загружаемого модуля был выбран язык программирования C, так как этот язык является наиболее широко используемый средством написания загружаемых модулей ядра. Для разработки пользовательского приложения был выбран язык C++. Данный выбор обусловлен следующими причинами:

1. Статическая типизация;
2. Сочетание высокоуровневых и низкоуровневых средств;
3. Богатая стандартная библиотека шаблонов;
4. Поддержка ООП.

Также для создания приложения была использована библиотека Qt, так как она имеет большой набор средств для создания приложений с графическим интерфейсом.

3.2 Пользовательское приложение

Для корректной работы разработанной программы мониторинга ОС linux необходимо аппаратное обеспечение на базе операционной системы Ubuntu версии 16.04, версия ядра 4.15.0-128-generic.

3.3 Описание интерфейса программы

Пользовательское приложение представляет из себя четыре виджета для отображения информации о системе и об информации, поступающей из пространства ядра. Также приложение предоставляет пользователю возможность задать ID того процесса, информацию о котором необходимо получить. Интерфейс приложения представлен на рисунках 11-12.

На первой вкладке представлена информация о системе. Данные обновляются каждую секунду.

Для того, чтобы получить дерево процессов, пользователю необходимо нажать на кнопку “Построить дерево процессов” вкладки “Дерево процессов”.

Для того, чтобы получить информацию о файлах и памяти процесса, пользователю необходимо ввести ID процесса во вкладках “Открытые файлы” и “Виртуальная память” соответственно, затем нажать кнопку “Получить информацию о файлах” или кнопку “Получить информацию о памяти” соответственно.

В случае, если процесс с заданным идентификатором не найден, приложение уведомит об этом пользователя посредством вывода сообщения. Также система сообщит пользователю, если он пытается запросить информацию о процессах в случае, когда модуль не загружен.

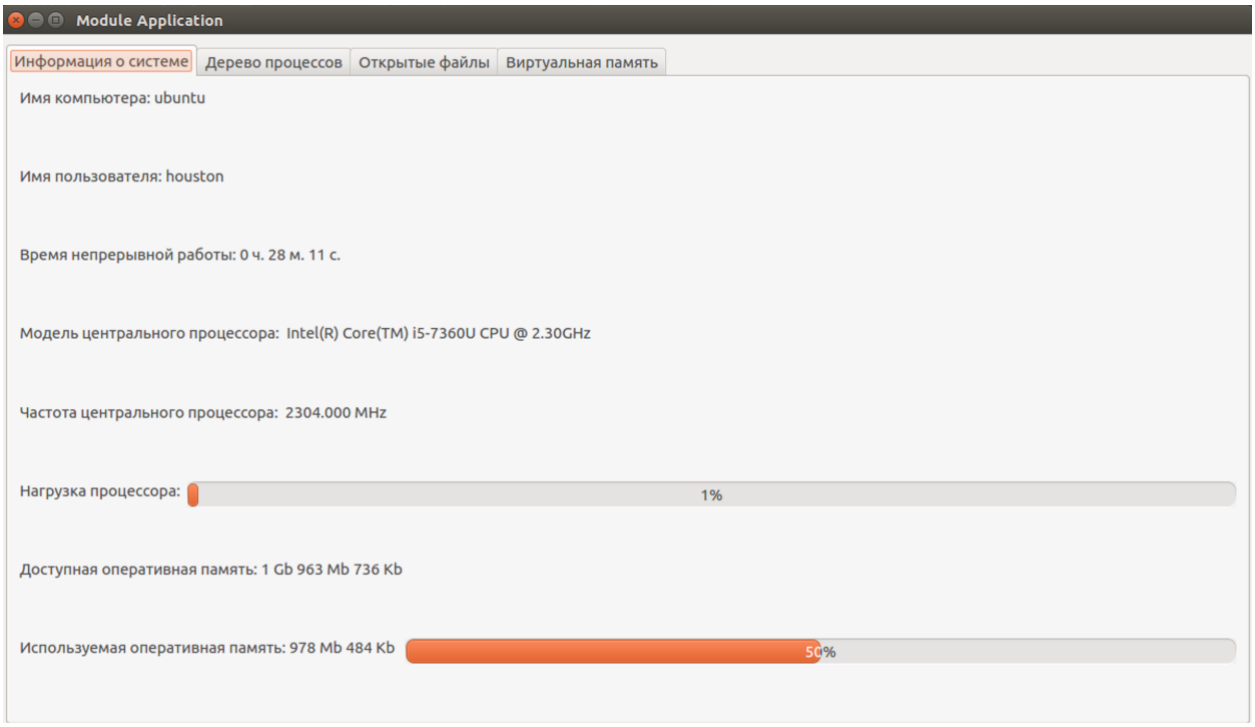

Рисунок 11. Интерфейс пользовательского приложения. Вкладка “Информация о системе”

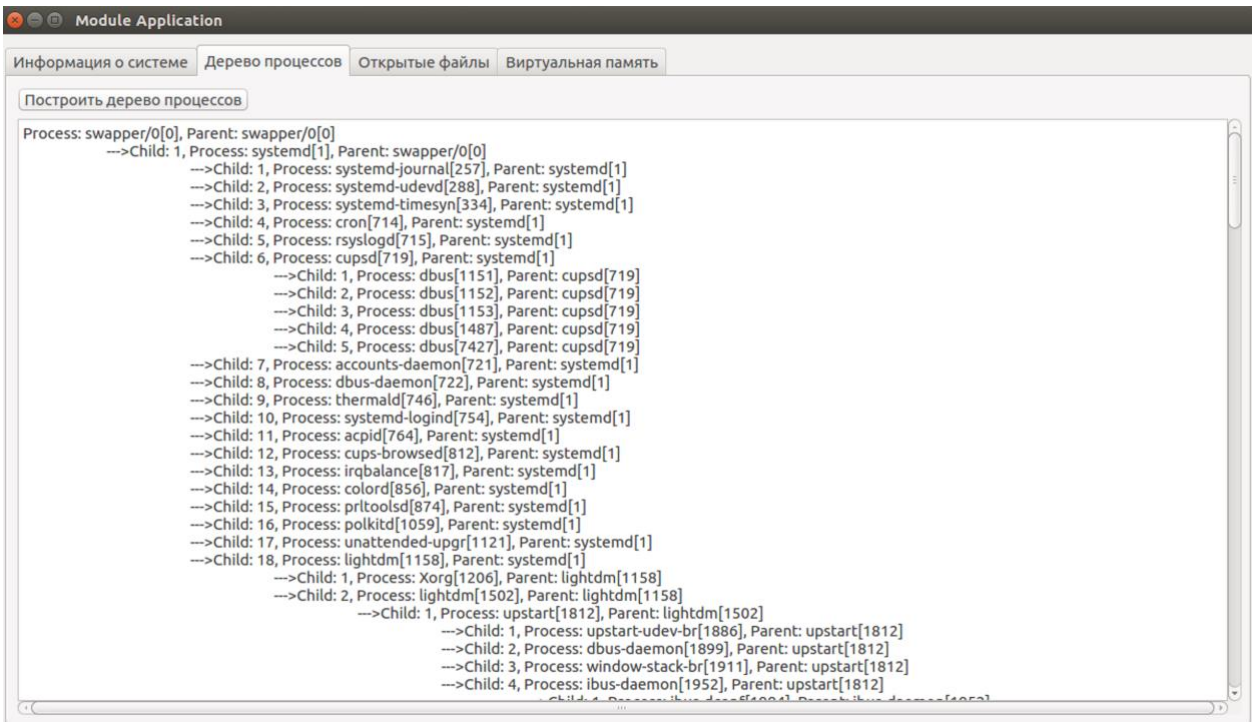

Рисунок 12. Интерфейс пользовательского приложения. Вкладка “Дерево процессов”

Вывод

В данном разделе были выбраны средства реализации, рассмотрен интерфейс программы

Заключение

В процессе работы были изучены основные подходы, применяемые для получения информации о процессах ОС Linux, принципы написания загружаемых модулей ядра. Также был изучен интерфейс файловой системы procfs, предоставляющий функционал для взаимодействия пространства ядра с пространством пользователя.

В результаты выполненной работы был разработан программный комплекс из загружаемого модуля ядра и пользовательского приложения, предоставляющий пользователю информацию о системе, дерево процессов, а также информацию о файлах и памяти указанного пользователем процесса.

Данный проект имеет перспективы развития, так как в будущем можно добавить в модуль новый функционал, связанный с выводом более широкого спектра информации о процессах системы. Также может быть улучшено и пользовательское приложение за счет изменения вывода информации в более удобном для пользователя виде. К примеру, данные о файлах и памяти могут быть представлены в виде таблиц.

Литература

1. Вахалия Ю. UNIX изнутри. Спб.: Питер, 2003.
2. Таненбаум Э., Бос Х. Современные операционные системы. . 4 изд. Спб.: Питер, 2015.
3. Анатомия загружаемых модулей ядра Linux // IBM developer URL: https://www.ibm.com/developerworks/ru/library/l-lkm/index.html (дата обращения: 04.12.2020).
4. sched.h // free-electronics URL: https://elixir.free-electrons.com/linux/latest/source/include/linux/sched.h (дата обращения: 14.12.2020).
5. fdtable // Kernel.org URL: https://www.kernel.org/doc/Documentation/filesystems/files.txt (дата обращения: 14.12.2020).
6. process // Kernel.org URL: https://www.kernel.org/doc/Documentation/process (дата обращения: 13.12.2020).

Приложение А

```
struct task_struct {
    volatile long state;
    void *stack;
    unsigned int flags;
     …
    int prio, static_prio;
     …
    struct list_head tasks;

    struct mm_struct *mm, *active_mm;
    …
    pid_t pid; pid_t tgid;
     …
    struct task_struct *real_parent;
     …
    char comm[TASK_COMM_LEN];
     …
    struct thread_struct thread;
    …
    struct files_struct *files;
    ...
};
```

Листинг 2. Фрагмент структуры task_struct

Приложение Б

```
#define    for_each_process(p)    for (p = &init_task ; (p = next_task(p)) != &init_task ; )

#define next_task(p)    list_entry((p)->tasks.next, struct task_struct, tasks)
```

Листинг 3. Определения макросов для перемещения по списку процессов

Приложение В

```
struct vm_area_struct {
struct mm_struct * vm_mm; /* параметры области виртуальной памяти */
unsigned long vm_start;
unsigned long vm_end;
/* Связянный список областей задачи отсортированный по адресам */
struct vm_area_struct *vm_next;
...
unsigned short vm_flags;
...
struct vm_operations_struct * vm_ops; /*операции над областью */
...
unsigned long vm_pte; /* разделяемая память */
};
```

Листинг 4. Фрагменты структуры vm_area_struct

Приложение Г

```
struct fdtable {
        unsigned int max_fds;
        struct file **fd;      /* current fd array */
        unsigned long *close_on_exec;
        unsigned long *open_fds;
        unsigned long *full_fds_bits;
        struct rcu_head rcu;
};
```

Листинг 5. Структура fdtable

Приложение Д

```
struct proc_dir_entry {
...
      const char *name;
      mode_t mode;
...
      uid_t uid;
      gid_t gid;
...
      const struct file_operations *proc_fops;
...
      read_proc_t *read_proc;
      write_proc_t *write_proc;
...
};
```

Листинг 6. Фрагмент структуры proc_dir_entry