Реферат

В данном курсовом проекте представлена файловая система для просмотра информации и управления удалёнными компьютерами. Файловая система реализована в виде модуля ядра Linux. Файловая система не связана с физическими устройствами хранения файлов и все файлы в ней создаются «на лету».

Оглавление

Введение

Одним из базовых принципов Unix гласит: «Всё является файлом». Сейчас «всё» действительно означает всё. Каталог, жесткий диск, раздел на жестком диске, параллельный порт, подключение к веб-сайту, карта Ethernet — всё это файлы. Linux, унаследовавший этот и многие другие принципы Unix, различает много типов файлов в дополнение к стандартным файлам и каталогам.

В ядре Linux помимо дисковых файловых систем (наиболее используемыми из которых являются `btrfs`, `ext3`, `ext4`, `ReiserFS`, `XFS`) присутствуют также системы, предоставляющие доступ к ресурсам компьютера, отличным от данных на жёстком диске. `procfs` позволяет получить доступ к информации о системных процессах из ядра (она необходима для выполнения таких команд как `ps`, `w`, `top`), `sysfs` экспортирует в пространство пользователя информацию ядра Linux о присутствующих в системе устройствах и драйверах, `ramfs` используется для создания RAM-диска в процессе загрузки системы, `tmpfs` поддерживает работу с виртуальной памятью. Для Linux также существуют сетевые системы: `SMB` (реализующая поддержку «общих папок Windows»), `NFS` (популярный в Unix-подобных системах протокол сетевого доступа к файловым системам), а также кластерные системы `Lustre`, `Cerp`, `GlusterFS` и другие. Реализованы также системы для обеспечения абстракции над сетевыми ресурсами: `sshfs` (доступ к файлам по протоколу `SSH`), `ftpfs` (доступ к файлам по протоколу `FTP`).

Не существует широко известных файловых систем для просмотра информации и удалённого управления компьютерами. В связи с этим было решено расширить принцип «всё есть файл», создав файловую систему, в которой файлами являлись бы параметры удалённых компьютеров и действия над ними.

1. Аналитический раздел

1.1. Метод реализации файловой системы

Существует три основных способа реализации файловой системы Linux [1].

1.1.1. Изменение исходного кода ядра

Изменение исходного кода ядра является наиболее затратным по временным ресурсам, поскольку при каждом изменении файловой системы необходимо пересобирать ядро. Также этот вариант не позволяет выпускать новые версии файловой системы чаще, чем новые версии ядра. К тому же файловая система для удалённого управления скорее всего не нужна рядовому пользователю и увеличит и так «раздутое и огромное» (Линус Торвальдс, 2009 г.) ядро.

1.1.2. Создание загружемого модуля ядра

Файловая система оформляется в виде загружаемого модуля ядра, компилируемого в бинарный файл формата kernel object (.ko), который загружается и выгружается командами `insmod` и `rmmod`.

Этот подход позволяет вести разработку быстрее (сборка модуля ядра занимает на порядки меньше времени, чем сборка всего ядра) и не зависеть от выпусков ядра. Недостатком являются ограниченные возможности по доступу к объектам ядра [2].

1.1.3. Использование FUSE

`FUSE` — загружаемый модуль для Unix-подобных ОС, который позволяет пользователям без привилегий создавать их собственные файловые системы без необходимости переписывать код ядра. Это достигается за счёт запуска кода файловой системы в пространстве пользователя, в то время как модуль `FUSE` только предоставляет мост для актуальных интерфейсов ядра. `FUSE` была официально включена в главное дерево кода Linux в версии 2.6.14.

К недостаткам этого подхода относится то, что предоставляя унифицированный интерфейс к созданию файловых систем многих ОС, `FUSE` не предоставляет доступа к низкоуровневым объектам реализации слоя файловой системы (VFS в Linux), что сказывается на производительности.

1.1.4. Выбор метода реализации файловой системы

Управление большим количеством удалённых компьютеров должно производиться максимально быстро. Поэтому было решено отказаться от использования `FUSE`. Так как доступ к объектам ядра, возможный только из самого ядра, не требовался, было решено реализовать файловую систему в виде модуля ядра.

1.2. Метод удалённого управления

Программирование в пространстве ядра накладывает существенные ограничения. Использование стандартной библиотеки C и библиотек GNU не представляется возможным, поэтому использование сокетов Беркли для сетевого взаимодействия является трудновыполнимой задачей. Поэтому было принято решение осуществлять сетевое взаимодействие в пространстве пользователя и лишь затем передавать данные в пространство ядра для обработки модулем файловой системы.

Существует большое количество ПО удалённого управления. В связи с этим представляется более разумным не создавать ещё одно ПО для удалённого управления, а приспособить уже имеющееся. Категория ПО, ориентированная на массовое управление серверами, отображение и изменение их конфигурации, называется «ПО для конфигурационного управления» (англ. configuration management software). Именно такое ПО отвечает требованиям, поставленным перед файловой системой.

Подробное сравнение свободного ПО для конфигурационного управления проведено в [3]. В связи со свободной лицензией (Apache),

поддержкой большого количества ОС (включая GNU/Linux, OS X и Microsoft Windows), а также скоростью работы, обусловленной использованием сетевой библиотеки ZeroMQ и протокола обмена сообщениями MessagePack, ориентированных на производительность, был выбран SaltStack (сокращённо — Salt). В связи с этим файловой системе было дано название `saltfs`.

2. Конструкторский раздел

2.1. Использование Salt

Зёрно (англ. grain) — статическая (меняющаяся не чаще, чем несколько раз, во время работы ОС) информация о компьютере, ОС и конфигурации.

Функция — действие, которое возможно осуществить над клиентским компьютером. Примеры: `test.ping` (проверка наличие миньона в сети), `system.reboot` (перезагрузка ОС), `pkg.install NAME` (установка пакета из репозитория).

Мастер (англ. master) — серверный компьютер, на котором запущена программа `salt-master`. Обычно в системе Salt находится один мастер.

Миньон (англ. minion) — клиентский компьютер, на котором запущена программа `salt-minion`, управляемый мастером. В системе Salt может находится произвольное количество миньонов (в наиболее крупных системах Salt находятся десятки тысяч миньонов).

Модуль — группа функций, схожих по смыслу. Примеры: `test` (модуль с функциями проверки работы системы Salt), `system` (модуль управления питания компьютера и основными параметрами ОС), `pkg` (модуль для управления установленным ПО).

2.2. Иерархия файловой системы

На рисунке 2.1 изображена иерархия файловой системы.

2.2.1. Первый уровень

На верхнем уровне находятся директории, именованные идентификаторами миньонов.

2.2.2. Второй уровень

На втором уровне находятся директории, соответствующие модулям (в общем случае, они различны для разных платформ: зависят от установленной ОС (например, модуль `linux_acl` для GNU/Linux)

и прикладного ПО (модули `postgres` и `mysql` присутствуют при установленных PostgreSQL и MySQL).

2.2.3. Третий уровень

На третьем уровне находятся файлы, соответствующие функциям. Для каждой функции определены операции чтения и записи. При чтении происходит выдача документации по заданной функции. Запись параметров функции служит для вызова функции.

Для управления гранулами в Salt служит модуль `grains`. Он обрабатывается особым образом. В нём, в отличие от определённых в Salt функций, содержатся файлы, соответствующие гранулам.

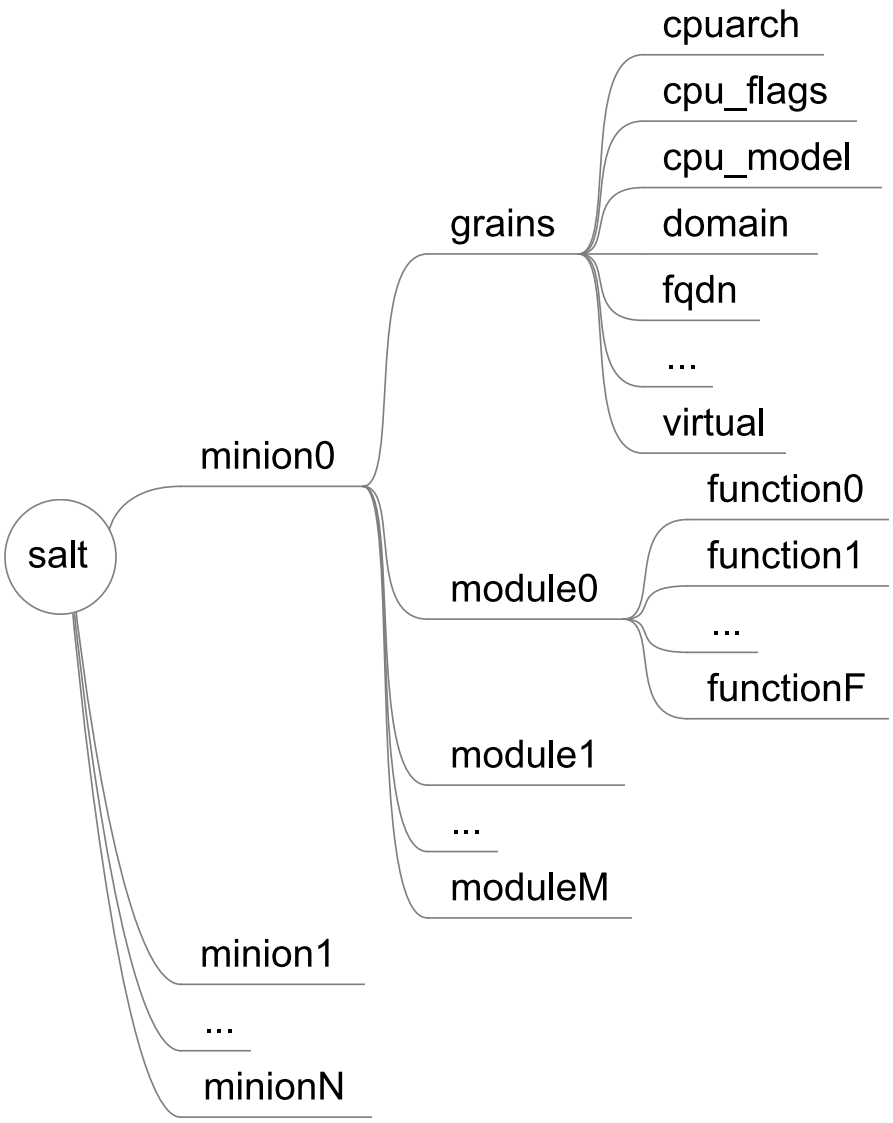

Рис. 2.1 — Иерархия файловой системы

2.3. Устройство файловой системы

Файловая система `saltfs` не связана с физическими устройствами хранения файлов и все файлы в ней создаются «на лету». Для получения списка директорий, содержимого файлов, вызова функций используется скрипт командной оболочки `Fish`, который вызывает программу `salt`.

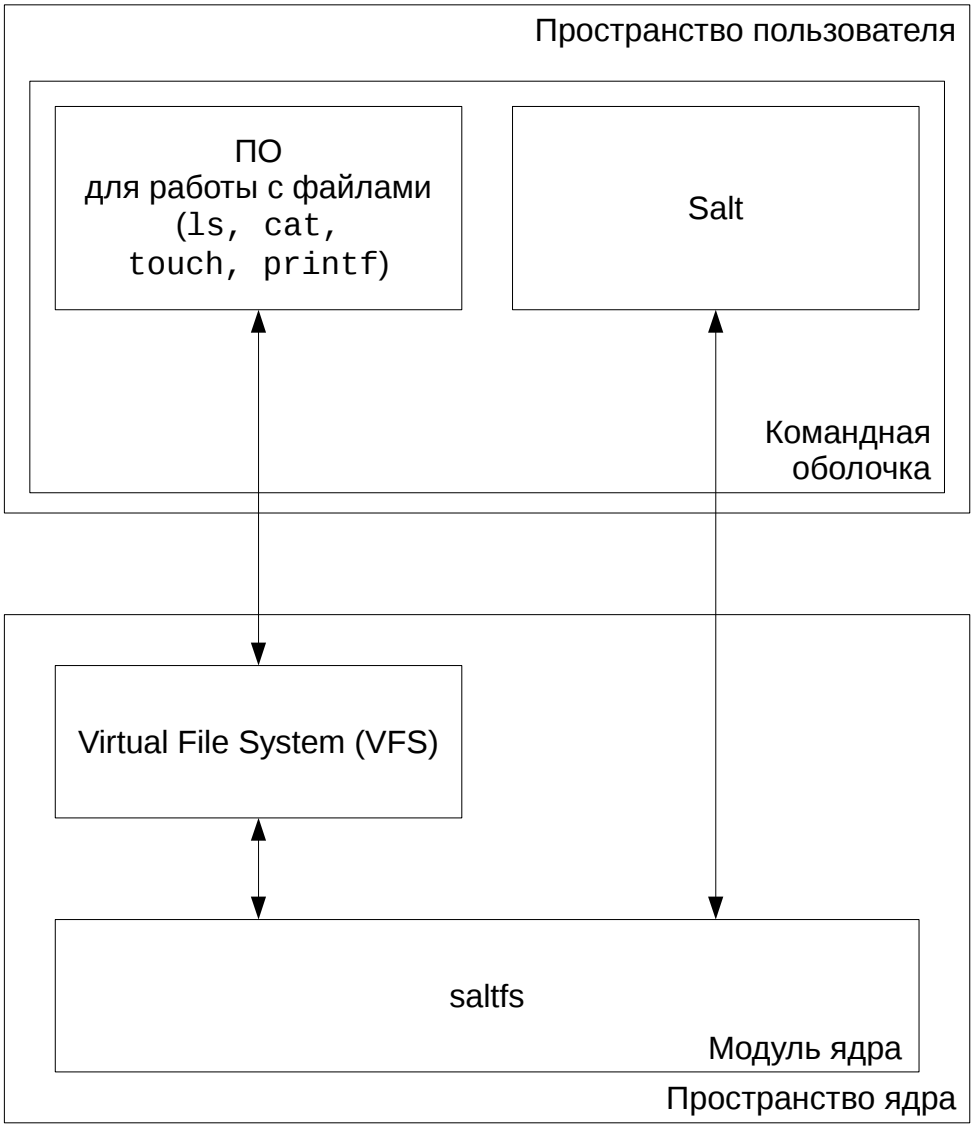

Рис. 2.2 — Взаимодействие файловой системы с ПО

2.4. Взаимодействие пространства ядра и пользователя

При выборе механизма взаимодействия модуля ядра `saltfs` с Salt использовался список способов взаимодействия [4]. Перечислим

основные достоинства и недостатки механизмов (применительно к конкретной задачи), рассмотренных в нём.

2.4.1. procfs

2.4.1.1. Достоинства

а) Простое создание объектов обмена данными.

б) Может служить для обмена произвольным количеством данных.

2.4.1.2. Недостатки

а) Считается устаревшим способом взаимодействия.

б) Директория `/proc` плохо структурирована.

в) Чтение из пространства пользователя происходит по байтам.

2.4.2. sysfs

2.4.2.1. Достоинства

а) Хорошее структурирование.

б) Стандартный способ взаимодействие с драйверами устройств.

2.4.2.2. Недостатки

а) Фиксированный размер файлов, ограниченный размером страницы `PAGE_SIZE` (4096 байт на i386).

2.4.3. configfs

2.4.3.1. Недостатки

а) Фиксированный размер файлов, ограниченный размером страницы `PAGE_SIZE` (4096 байт на i386).

б) Создание объектов происходит из пространства пользователя.

2.4.4. debugfs

2.4.4.1. Достоинства

а) Простота создания объектов обмена данными.

б) Простота чтения и записи.

2.4.4.2. Недостатки

а) В стандартной поставке ядра Linux данная ФС отключена.

2.4.5. sysctl

2.4.5.1. Недостатки

а) Используется для настройки параметров ядра во время его работы.

б) Возможна запись только чисел.

2.4.6. Символьные устройства

2.4.6.1. Достоинства

а) Может служить для обмена произвольным количеством данных.

2.4.6.2. Недостатки

а) Пользователь должен создавать устройства самостоятельно.

2.4.7. Netlink сокеты

2.4.7.1. Достоинства

а) Предоставляют большую гибкость в использовании, чем способы взаимодействия, основанные на файлах.

б) Объекты обмена данными не видны пользователю.

2.4.7.2. Недостатки

а) Большая сложность использования, чем у способов взаимодействия, основанных на файлах.

б) Необходимость написания специального приложения для взаимодействия в пространстве пользователя.

2.4.8. Системные вызовы ядра

2.4.8.1. Достоинства

а) Возможность получения текстовой информации в виде строки-аргумента вызова.

2.4.8.2. Недостатки

а) Требует модификации ядра.

2.4.9. Mmap

2.4.9.1. Достоинства

а) Не требует копирования между пространством ядра и пространством пользователя.

б) Может служить для обмена произвольным количеством данных.

2.4.9.2. Недостатки

а) Необходимость в использовании объектов синхронизации для оповещения о поступлении новых данных.

б) Необходимость написания специального приложения для взаимодействия в пространстве пользователя.

2.5. Обоснование выбора

В качестве механизма взаимодействия был выбран обмен данными с помощью `procfs` из-за простоты его использования, отсутствии необходимости написания дополнительного пользовательского

приложения и отсутствии недостатков, не позволяющих его использовать.

В качестве способ запуска программы `salt` было использовано API usermode_helper [5].

3. Технологический раздел

3.1. Выбор языка и среды разработки

Ядро предоставляет возможность написания модулей лишь на языке C. Пользовательская часть (скрипт вызова `salt`, тесты) была написана на языке командной оболочки Fish, поскольку её синтаксис отличается от Bash повышенной читаемостью и простотой сопровождения.

В качестве среды разработки использовалось IDE CLion.

3.2. Выбор системы виртуализации

В качестве системы виртуализации была выбрана связка VirtualBox + Vagrant. VirtualBox является наиболее популярной пользовательской системой виртуализации, а Vagrant позволяет автоматизировать процесс разработки, позволяя свести процесс перезагрузки виртуальной машины и загрузки модуля к запуску заранее написанного скрипта.

3.3. Выбор дистрибутива GNU/Linux и ядра

В качестве дистрибутива Linux был использован openSUSE 13.2. В качестве ядра было использовано стандартное ядро версии 3.16, идущее в комплекте.

3.4. Особенности реализации

При разработке многих функций файловой системы использовалась библиотека ядра для разработчиков файловых систем libfs [6].

Разработка велась по стандарту кодирования, описанному в документации ядра Linux.

Основным источником документации по созданию файловых систем являются сами исходные коды ядра Linux [7]. В основном были использованы коды файловых систем `procfs`, `logfs`, `ramfs`.

3.5. Получение списка файлов в директории

На рисунке 3.1 и 3.2 приведена схема алгоритма получения списка файлов в директории.

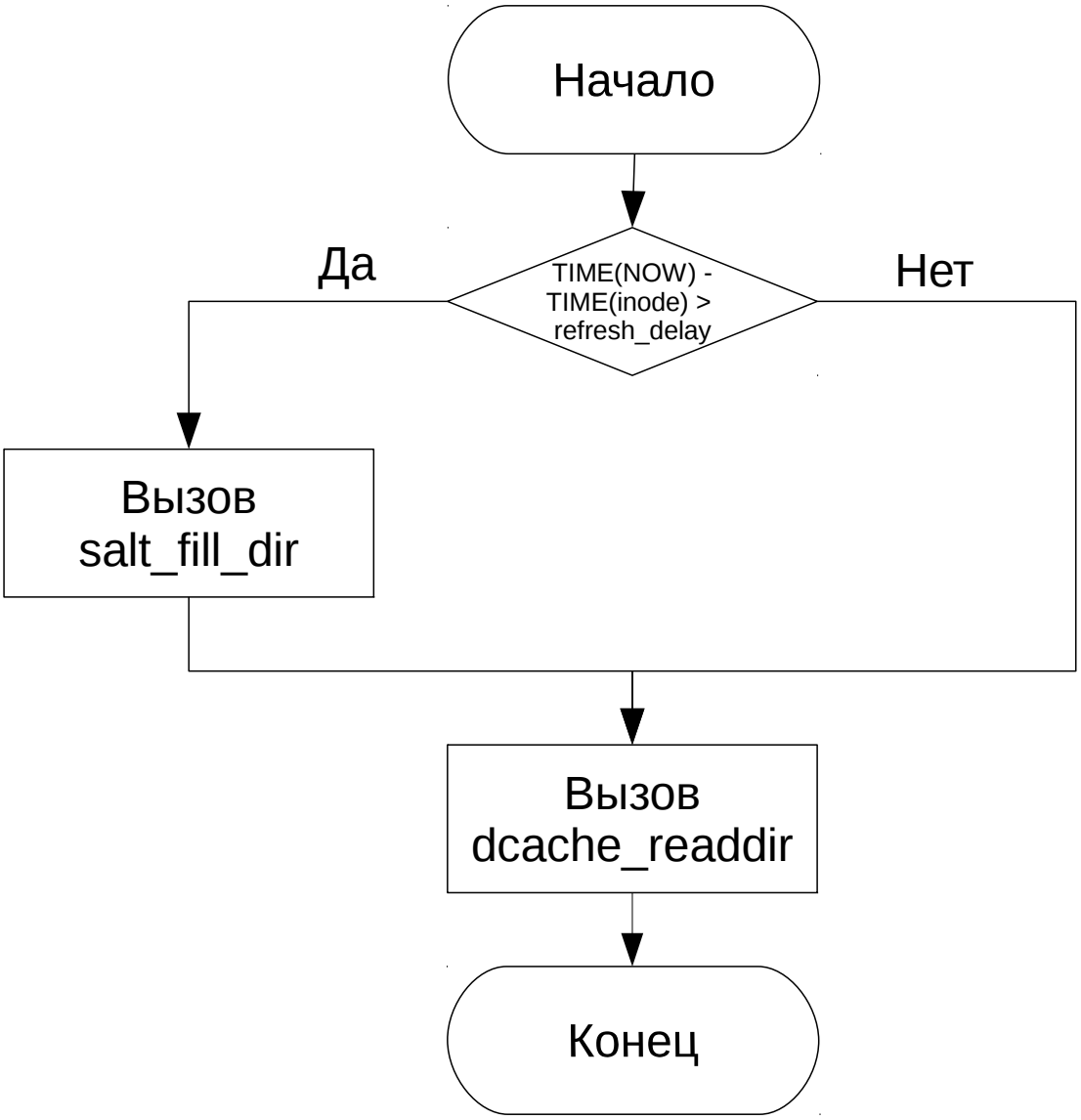

Рис. 3.1 — Схема алгоритма получения списка файлов в директории

Рис. 3.2 — Схема алгоритма получения списка файлов в директории

Заключение

В данной работе была показана актуальность создания файловой системы для просмотра информации и управления удалёнными компьютерами. Требования, поставленные в техническом задании, были выполнены.

В процессе работы над курсовым проектом была изучены существующие файловые системы Linux, процесс написания модуля ядра, структуры данных ядра, способы взаимодействия приложений пространства пользователя с модулем ядра. Также были получены практические навыки использования средств виртуализации и систем конфигурационного управления.

Литература

1. *Robert Love.* Linux kernel development / Robert Love. — Boston: Addison-Wesley, 2010.

2. *Peter Jay Salzman, Michael Burian, Ori Pomerantz,* . The Linux Kernel Module Programming Guide / Peter Jay Salzman, Michael Burian, Ori Pomerantz, . — CreateSpace, 2005.

3. *Wikipedia, Авторы.* Comparison of open-source configuration management software. — 2014. — декабрь. `https://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software`.

4. *Keller, Ariane.* — Kernel Space - User Space Interfaces, 2008. — июль. `http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html`.

5. *Jones, M. Tim.* Invoking user-space applications from the kernel / M. Tim Jones // *Kernel APIs.* — 2010. — no. 1. `http://www.ibm.com/developerworks/library/l-user-space-apps/`.

6. *corbet.* Creating Linux virtual filesystems / corbet // *LWN.net.* — 2003. — ноябрь. `http://lwn.net/Articles/57369/`.

7. *Experts, Embedded Linux.* Linux Cross Reference. — 2014. `http://lxr.free-electrons.com/`.

Приложение А. Исходные коды

```
#include "dir.h"
#include "function.h"
#include "grain.h"
#include "result.h"
#include "string.h"
#include "user.h"

#include <linux/idr.h>
#include <linux/slab.h>
#include <linux/types.h>

#define SALT_OUTPUT_DIR /proc/
#define SALT_OUTPUT_FILE "/proc/saltfs"


static int const refresh_delay = 4;  /* in seconds */

extern struct salt_item_spec const salt_items_spec[];

struct salt_dir_entry const *parent(struct salt_dir_entry const *sde,
        enum salt_dir_entry_type const type)
{
    while (sde && sde->type != type)
        sde = sde->parent;
    return sde;
}

static char *list_cmd_root(struct salt_dir_entry const *sde) {
    pr_debug("saltfs: get list_cmd_root string\n");
    return vstrcat("", NULL);
}

static char *list_cmd_minion(struct salt_dir_entry const *sde) {
    pr_debug("saltfs: get list_cmd_minion string\n");
    return vstrcat("__fish_salt_list_minion accepted", NULL);
}

static char *list_cmd_module(struct salt_dir_entry const *sde) {
    char const *minion = parent(sde, Salt_minion)->name;
    pr_debug("saltfs: get list_cmd_module string; minion=%s\n", minion);
    return vstrcat(SALT_FISH_SET_MINION(minion),
            "__fish_salt_list_module", NULL);
}

static char *list_cmd_function(struct salt_dir_entry const *sde) {
```

```
46      char const *minion = parent(sde, Salt_minion)->name,
47              *module = parent(sde, Salt_module)->name;
48      pr_debug("saltfs: get list_cmd_funtion string; minion=%s, module=%s\n",
49              minion, module);
50      return vstrcat(SALT_FISH_SET_MINION(minion),
51              "__fish_salt_list_function_without_module ", module, NULL);
52  }
53
54  static char *list_cmd_grain(struct salt_dir_entry const *sde) {
55      char *minion = sde->parent->name;
56      pr_debug("saltfs: get list_cmd_grain string; minion=%s\n", minion);
57      return vstrcat(SALT_FISH_SET_MINION(minion),
58              "__fish_salt_list_grain", NULL);
59  }
60
61  struct salt_next_item_spec const salt_module_next_items[] = {
62          { .name = "grains", .type = Salt_grain,    },
63          {                   .type = Salt_function, },
64  };
65
66  struct salt_item_spec const salt_items_spec[] = {
67          {
68                  .name = "NULL",
69          },
70          {
71                  .name = "root",
72                  .list_cmd = list_cmd_root,
73                  .next_item_type = Salt_minion,
74                  .mode = S_IFDIR,
75          },
76          {
77                  .name = "minion",
78                  .list_cmd = list_cmd_minion,
79                  .next_item_type = Salt_module,
80                  .mode = S_IFDIR,
81          },
82          {
83                  .name = "module",
84                  .list_cmd = list_cmd_module,
85                  .next_item_type = Salt_function,
86                  .next_items = salt_module_next_items,
87                  .mode = S_IFDIR,
88          },
89          {
90                  .name = "function",
91                  .list_cmd = list_cmd_function,
```

```
92                  .fops = &salt_function_fops,
93                  .mode = S_IFREG,
94          },
95          {
96                  .name = "grain",
97                  .list_cmd = list_cmd_grain,
98                  .fops = &salt_grain_fops,
99                  .mode = S_IFREG,
100         },
101         {
102                 .name = "result",
103                 .fops = &salt_result_fops,
104                 .mode = S_IFREG,
105         }
106 };
107
108
109 void salt_dir_entry_create(struct inode *dir, char const *name,
110         enum salt_dir_entry_type const type, struct inode *parent)
111 {
112     unsigned int len = strlen(name);
113     struct salt_dir_entry *sde =
114             kzalloc(sizeof(struct salt_dir_entry) + len + 1, GFP_KERNEL);
115
116     pr_debug("saltfs: salt_fill_salt_dir_entry: variables inited\n");
117
118     memcpy(sde->name, name, len + 1);
119     sde->type = type;
120     sde->parent = SDE(parent);
121     SALT_I(dir)->sde = sde;
122     pr_debug("saltfs: new salt_inode filled; i_ino=%zu, name='%s', type=%d\n",
123             dir->i_ino, sde->name, sde->type);
124 }
125
126 void salt_dir_entry_free(struct salt_dir_entry *sde)
127 {
128     pr_debug("saltfs: free '%s'\n", sde->name);
129     kfree(sde);
130 }
131
132 static int dentry_delete(struct dentry const *de)
133 {
134     pr_debug("saltfs: dentry_delete '%s'\n", SDE(de->d_inode)->name);
135     salt_dir_entry_free(SDE(de->d_inode));
136     return 1;
137 }
```

```
138
139 static void dentry_release(struct dentry *de)
140 {
141     pr_debug("saltfs: dentry_release '%s'\n", SDE(de->d_inode)->name);
142     salt_dir_entry_free(SDE(de->d_inode));
143 }
144
145 static void dentry_prune(struct dentry *de)
146 {
147     pr_debug("saltfs: dentry_prune '%s'\n", SDE(de->d_inode)->name);
148     salt_dir_entry_free(SDE(de->d_inode));
149 }
150
151 struct dentry_operations const salt_dentry_operations = {
152         .d_delete = dentry_delete,
153         .d_release = dentry_release,
154         .d_prune = dentry_prune,
155 };
156
157 void update_current_time(struct inode *inode)
158 {
159     inode->i_atime = (struct timespec){0, 0};
160     inode->i_mtime = inode->i_ctime = CURRENT_TIME;
161 }
162
163 static struct inode *salt_inode_create(struct inode *dir, struct dentry *dentry,
164         enum salt_dir_entry_type const type)
165 {
166     struct inode *inode;
167     umode_t mode = salt_items_spec[type].mode;
168
169     inode = new_inode(dir->i_sb);
170
171     pr_debug("saltfs: new inode created\n");
172
173     if (!inode)
174         return inode;
175
176     inode->i_ino = get_next_ino();
177     inode->i_op = &simple_dir_inode_operations;
178     inode->i_fop = (salt_items_spec[type].fops)?
179             salt_items_spec[type].fops : &salt_dir_operations;
180     inode_init_owner(inode, NULL, mode | 0770);
181     switch (mode & S_IFMT) {
182         case S_IFREG:
183             inode->i_flags |= O_RDWR | O_CREAT;
```

```
184             break;
185         case S_IFDIR:
186             inode->i_flags |= S_IMMUTABLE;
187             inc_nlink(inode);
188             break;
189     }
190     update_current_time(inode);
191 
192     pr_debug("saltfs: new inode filled; "
193                     "type=%d, i_ino=%zu, i_mode=%d, i_flags=%ul\n",
194             type, inode->i_ino, inode->i_mode, inode->i_flags);
195 
196     return inode;
197 }
198 
199 bool salt_fill_cache_item(struct dentry *dir, char const *name, int len,
200         enum salt_dir_entry_type const type)
201 {
202     struct dentry *child;
203     struct qstr qname = QSTR_INIT(name, len);
204     struct inode *inode, *parent_inode = dir->d_inode;
205 
206     pr_debug("saltfs: salt_fill_cache_item: name='%s', i_ino: %zu\n",
207             name, parent_inode->i_ino);
208 
209     child = d_hash_and_lookup(dir, &qname);
210     if (!child) {
211         child = d_alloc(dir, &qname);
212         pr_debug("saltfs: salt_fill_cache_item: allocated child\n");
213 
214         inode = salt_inode_create(parent_inode, dir, type);
215         salt_dir_entry_create(inode, name, type, parent_inode);
216 
217         d_add(child, inode);
218         d_set_d_op(child, &salt_dentry_operations);
219         pr_debug("saltfs: new dentry cached\n");
220     }
221 
222     return 0;
223 }
224 
225 void salt_fill_dir(struct salt_dir_entry *sde, struct dentry *dir, int const ino,
226         enum salt_dir_entry_type const type)
227 {
228     int i = 0;
229     char *next_item_list_cmd;
```

```
230     char *salt_item;
231     struct salt_userspace_output const *salt_output;
232     struct salt_next_item_spec const *next_item =
233             salt_items_spec[type].next_items;
234     enum salt_dir_entry_type next_item_type;
235
236     if (next_item) {
237         while (next_item->name) {
238             if (strcmp(next_item->name, sde->name) == 0)
239                 break;
240             next_item++;
241         }
242         next_item_type = next_item->type;
243         pr_debug("saltfs: salt_readdir: next_item: name='%s', type=%d\n",
244                 next_item->name, next_item->type);
245     } else {
246         next_item_type = salt_items_spec[type].next_item_type;
247     }
248
249     next_item_list_cmd = salt_items_spec[next_item_type].list_cmd(sde);
250     pr_debug("saltfs: salt_readdir: next_item: list_cmd='%s'\n",
251             next_item_list_cmd);
252
253     salt_list(next_item_list_cmd, ino);
254     kfree(next_item_list_cmd);
255
256     salt_output = (struct salt_userspace_output const *)
257             idr_find(&salt_output_idr, ino);
258     for (i = 0; i < salt_output->line_count; i++) {
259         salt_item = salt_output->lines[i];
260         salt_fill_cache_item(dir, salt_item, strlen(salt_item), next_item_type);
261     }
262 }
263
264 static int salt_dir_iterate(struct file *file, struct dir_context *ctx)
265 {
266     struct inode *inode = file->f_inode;
267     struct salt_dir_entry *sde = SDE(inode);
268     enum salt_dir_entry_type next_item_type =
269             salt_items_spec[sde->type].next_item_type;
270
271 #ifdef DEBUG
272     int const path_len = 1024;
273     char buf[path_len];
274     char *path;
275
```

```
276     path = dentry_path_raw(file->f_path.dentry, buf, path_len - 1);
277     pr_debug("saltfs: salt_readdir: called with dir '%s', type %d, i_ino=%zu\n",
278             path, sde->type, inode->i_ino);
279 #endif
280 
281     pr_debug("saltfs: salt_readdir: next_item: name='%s', type=%d\n",
282             salt_items_spec[next_item_type].name, next_item_type);
283 
284     /* Need to make more intelligent cache invalidation */
285     if (CURRENT_TIME.tv_sec - inode->i_atime.tv_sec > refresh_delay) {
286         pr_debug("saltfs: salt_readdir: cache invalidation\n");
287         inode->i_atime = CURRENT_TIME;
288         salt_fill_dir(sde, file->f_path.dentry, inode->i_ino, sde->type);
289     }
290 
291     return dcache_readdir(file, ctx);
292 }
293 
294 struct file_operations const salt_dir_operations = {
295         .open    = dcache_dir_open,
296         .llseek  = generic_file_llseek,
297         .read    = generic_read_dir,
298         .iterate = salt_dir_iterate,
299 };
```

Листинг 1 — dir.c

```
1  #ifndef __SALTFS_DIR_H__
2  #define __SALTFS_DIR_H__
3  
4  #include "internal.h"
5  
6  #include <linux/fs.h>
7  
8  struct salt_next_item_spec {
9      enum salt_dir_entry_type const type;
10     char const *name;
11 };
12 
13 struct salt_item_spec {
14     char *name;
15     char *(*list_cmd)(struct salt_dir_entry const *si);
16     struct file_operations const *fops;
17     enum salt_dir_entry_type next_item_type;
18     struct salt_next_item_spec const *next_items;
19     umode_t mode;
```

```
};

struct salt_dir_entry const *parent(struct salt_dir_entry const *sde,
        enum salt_dir_entry_type const type);
void update_current_time(struct inode *inode);
void salt_dir_entry_create(struct inode *dir, char const *name,
        enum salt_dir_entry_type const type, struct inode *parent);
void salt_fill_dir(struct salt_dir_entry *sde, struct dentry *dir, int const ino,
        enum salt_dir_entry_type const type);
bool salt_fill_cache_item(struct dentry *dir, char const *name, int len,
        enum salt_dir_entry_type const type);

extern struct file_operations const salt_dir_operations;
extern struct dentry_operations const salt_dentry_operations;
extern struct salt_item_spec const salt_items_spec[];

#endif /*__SALTFS_DIR_H__*/
```

Листинг 2 — dir.h

```
#include "user.h"

#include <asm/uaccess.h>
#include <linux/idr.h>
#include <linux/seq_file.h>

int salt_output_show(struct seq_file *m,
        struct salt_userspace_output const *salt_output)
{
    int i;
    for (i = 0; i < salt_output->line_count; i++)
        seq_printf(m, "%s\n", salt_output->lines[i]);
    return 0;
}

int salt_output_show_ino(struct seq_file *m, int ino)
{
    return salt_output_show(m, (struct salt_userspace_output const *)
            idr_find(&salt_output_idr, ino));
}
```

Листинг 3 — file.c

```
#ifndef __SALTFS_FILE_H__
#define __SALTFS_FILE_H__
```

```
3
4  struct salt_userspace_output;
5  struct seq_file;
6
7  int salt_output_show(struct seq_file *m,
8          struct salt_userspace_output const *salt_output);
9  int salt_output_show_ino(struct seq_file *m, int ino);
10
11 #endif /*__SALTFS_FILE_H__*/
```

Листинг 4 — file.h

```
1  #include "dir.h"
2  #include "file.h"
3  #include "string.h"
4  #include "user.h"
5
6  #include <asm/uaccess.h>
7  #include <linux/idr.h>
8  #include <linux/seq_file.h>
9  #include <linux/slab.h>
10
11
12 int salt_last_result_ino = 0;
13
14 char const *function_name(struct salt_dir_entry const *sde)
15 {
16     return vstrcat(parent(sde, Salt_module)->name, ".",
17             parent(sde, Salt_function)->name, NULL);
18 }
19
20 int salt_function_call(char const *minion, char const *function,
21         char const *args, int const ino)
22 {
23     int i;
24     char *cmd = vstrcat("salt ", minion, " ", function, " ", args, NULL);
25     pr_debug("saltfs: showing function '%s', ino=%d, cmd='%s'\n",
26             function, ino, cmd);
27     salt_list(cmd, ino);
28     kfree(cmd);
29     salt_output = (struct salt_userspace_output *)
30             idr_find(&salt_output_idr, ino);
31     for (i = 0; i < salt_output->line_count; i++)
32         pr_info("saltfs: result: %s\n", salt_output->lines[i]);
33     salt_last_result_ino = ino;
34     return 0;
```

```
35  }
36
37  int salt_function_call_parent_minion(struct inode const *inode,
38          char const *function, char const *args)
39  {
40      char const *minion = parent(SDE(inode), Salt_minion)->name;
41      return salt_function_call(minion, function, args, inode->i_ino);
42  }
43
44  int salt_function_call_inode(struct inode const *inode, char const *args)
45  {
46      char const *function = function_name(SDE(inode));
47      int ret = salt_function_call_parent_minion(inode, function, args);
48      kfree(function);
49      return ret;
50  }
51
52  static ssize_t salt_function_write(struct file *file,
53          char const *buf, size_t count, loff_t *offset)
54  {
55      int i;
56      struct inode *inode = file->f_inode;
57      char *args = (char *)kcalloc(count + 1, sizeof(char), GFP_KERNEL);
58      for (i = 0; i < count; i++)
59          get_user(args[i], buf + i);
60      if (args[i] == '\n')
61          args[i] = '\0';
62      salt_function_call_inode(inode, args);
63      return count;
64  }
65
66  static int salt_function_show(struct seq_file *m, void *v)
67  {
68      struct inode *inode = (struct inode *)(m->private);
69      struct salt_dir_entry *sde = SDE(inode);
70      int ino = inode->i_ino;
71      char const *minion = parent(sde, Salt_minion)->name;
72      char const *function = function_name(sde);
73      char *cmd = vstrcat(SALT_FISH_SET_MINION(minion),
74              "salt ", minion, " sys.doc ", function, NULL);
75      pr_debug("saltfs: showing function doc '%s', ino=%d, cmd='%s'\n",
76              function, ino, cmd);
77      salt_list(cmd, ino);
78      kfree(function);
79      kfree(cmd);
80      salt_output_show_ino(m, ino);
```

```
81      return 0;
82  }
83
84  bool is_touch(struct file const *file)
85  {
86      return (file->f_flags & (O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK));
87  }
88
89  static int salt_function_open(struct inode *inode, struct file *file)
90  {
91      pr_debug("saltfs: open function '%s', flags=%d\n",
92              SDE(inode)->name, file->f_flags);
93      if (is_touch(file))
94          salt_function_call_inode(file->f_inode, NULL);
95      return single_open(file, salt_function_show, inode);
96  }
97
98  struct file_operations const salt_function_fops = {
99          .open    = salt_function_open,
100         .read    = seq_read,
101         .write   = salt_function_write,
102         .llseek     = seq_lseek,
103         .release = single_release,
104 };
```

Листинг 5 — function.c

```
1   #ifndef __SALTFS_FUNCTION_H__
2   #define __SALTFS_FUNCTION_H__
3
4   extern struct file_operations const salt_function_fops;
5   extern int salt_function_call(char const *minion, char const *function,
6           char const *args, int const ino);
7   extern int salt_function_call_parent_minion(struct inode const *inode,
8           char const *function, char const *args);
9   extern int salt_function_call_inode(struct inode const *inode, char const *args);
10
11  extern int salt_last_result_ino;
12
13  #endif /*__SALTFS_FUNCTION_H__*/
```

Листинг 6 — function.h

```
1   #include "dir.h"
2   #include "file.h"
```

```
#include "function.h"
#include "string.h"
#include "user.h"

#include <asm/uaccess.h>
#include <linux/seq_file.h>
#include <linux/slab.h>


static int salt_grain_show(struct seq_file *m, void *v)
{
    struct inode *inode = (struct inode *)(m->private);
    struct salt_dir_entry *sde = SDE(inode);
    int ino = inode->i_ino;
    char const *grain = parent(sde, Salt_grain)->name;
    char const *minion = parent(sde, Salt_minion)->name;
    char *cmd = vstrcat(SALT_FISH_SET_MINION(minion),
            "__fish_salt_grain_read ", grain, NULL);
    pr_debug("saltfs: showing grain '%s', ino=%d, cmd='%s'\n", grain, ino, cmd);
    salt_list(cmd, ino);
    kfree(cmd);
    salt_output_show_ino(m, ino);
    return 0;
}

static int salt_grain_open(struct inode *inode, struct file *file)
{
    pr_debug("saltfs: open grain '%s'\n", SDE(inode)->name);
    return single_open(file, salt_grain_show, (void *)inode);
}

static ssize_t salt_grain_write(struct file *file,
        char const *buf, size_t count, loff_t *offset)
{
    int i;
    struct inode *inode = file->f_inode;
    char const *key = SDE(inode)->name;
    char *args, *value = (char *)kcalloc(count + 1, sizeof(char), GFP_KERNEL);
    for (i = 0; i < count; i++)
        get_user(value[i], buf + i);
    if (value[i] == '\n')
        value[i] = '\0';
    args = vstrcat(key, " ", value, NULL);
    salt_function_call_parent_minion(inode, "grains.setval", args);
    kfree(args);
    return count;
```

```
}

struct file_operations const salt_grain_fops = {
        .open    = salt_grain_open,
        .read    = seq_read,
        .write   = salt_grain_write,
        .llseek  = seq_lseek,
        .release = single_release,
};
```

Листинг 7 — grain.c

```
#ifndef __SALTFS_GRAIN_H__
#define __SALTFS_GRAIN_H__

extern struct file_operations const salt_grain_fops;

#endif /*__SALTFS_GRAIN_H__*/
```

Листинг 8 — grain.h

```
#include "inode.h"
#include "internal.h"

#include <linux/slab.h>

static struct kmem_cache *salt_inode_cachep;

struct inode *salt_alloc_inode(struct super_block *sb)
{
    struct salt_inode *si;
    struct inode *inode;

    si = (struct salt_inode *)kmem_cache_alloc(salt_inode_cachep, GFP_KERNEL);
    if (!si)
        return NULL;
    si->sde = NULL;
    inode = &si->vfs_inode;
    inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
    return inode;
}

static void init_once(void *foo)
{
    struct salt_inode *ei = (struct salt_inode *)foo;
```

```
25      inode_init_once(&ei->vfs_inode);
26  }
27
28  void __init salt_init_inodecache(void)
29  {
30      salt_inode_cachep = kmem_cache_create(
31              "salt_inode_cache", sizeof(struct salt_inode), 0,
32              (SLAB_RECLAIM_ACCOUNT | SLAB_MEM_SPREAD | SLAB_PANIC), init_once
33      );
34  }
```

Листинг 9 — inode.c

```
1   #ifndef __SALTFS_INTERNAL_H__
2   #define __SALTFS_INTERNAL_H__
3
4   #include <linux/proc_fs.h>
5
6   #define FS_NAME "saltfs"
7   /* only root should be able to access saltfs */
8   #define SALTFS_DEFAULT_MODE 0700
9   #define SALT_ROOT_INO 1
10
11  enum salt_dir_entry_type {
12      Salt_TYPE_NULL = 0,
13      Salt_root = 1,
14      Salt_minion = 2,
15      Salt_module = 3,
16      Salt_function = 4,
17      Salt_grain = 5,
18      Salt_result = 6,
19      Salt_TYPE_LAST = 7,
20  };
21
22  struct salt_dir_entry {
23      enum salt_dir_entry_type type;
24      struct salt_dir_entry *parent;
25      char name[];
26  };
27
28  struct salt_inode {
29      struct salt_dir_entry *sde;
30      struct inode vfs_inode;
31  };
32
33  static inline struct salt_inode *SALT_I(const struct inode *inode)
```

```
34 {
35     return container_of(inode, struct salt_inode, vfs_inode);
36 }
37
38 static inline struct salt_dir_entry *SDE(const struct inode *inode)
39 {
40     return (inode)? SALT_I(inode)->sde : NULL;
41
42 }
43
44 #endif /*__SALTFS_INTERNAL_H__*/
```

Листинг 10 — internal.h

```
1  #include "dir.h"
2  #include "file.h"
3  #include "function.h"
4  #include "string.h"
5
6  #include <asm/uaccess.h>
7  #include <linux/seq_file.h>
8  #include <linux/slab.h>
9
10 extern int salt_last_result_ino;
11
12 static int salt_result_show(struct seq_file *m, void *v)
13 {
14     if (salt_last_result_ino) {
15         pr_debug("saltfs: showing result for inode=%d\n", salt_last_result_ino);
16         return salt_output_show_ino(m, salt_last_result_ino);
17     }
18     return EPERM;
19 }
20
21 static int salt_result_open(struct inode *inode, struct file *file)
22 {
23     pr_debug("saltfs: open result\n");
24     return single_open(file, salt_result_show, (void *)inode);
25 }
26
27 struct file_operations const salt_result_fops = {
28         .open    = salt_result_open,
29         .read    = seq_read,
30         .llseek  = seq_lseek,
```

```
31          .release = single_release,
32  };
```

Листинг 11 — result.c

```
1   #ifndef __SALTFS_RESULT_H__
2   #define __SALTFS_RESULT_H__
3
4   extern struct file_operations const salt_result_fops;
5
6   #endif /*__SALTFS_RESULT_H__*/
```

Листинг 12 — result.h

```
1   #include "string.h"
2
3   #include <linux/printk.h>
4   #include <linux/slab.h>
5   #include <linux/string.h>
6
7   char *vstrcat(char const *first, ...)
8   {
9       size_t len;
10      char *retbuf;
11      va_list argp;
12      char *p;
13
14      if(first == NULL)
15          return NULL;
16
17      len = strlen(first);
18
19      va_start(argp, first);
20
21      while((p = va_arg(argp, char *)) != NULL)
22          len += strlen(p);
23
24      va_end(argp);
25
26      retbuf = kmalloc(len + 1, GFP_KERNEL);
27
28      if (retbuf == NULL)
29          return NULL;                            /* error */
30
31      (void)strcpy(retbuf, first);
```

```
32
33      va_start(argp, first);                          /* restart; 2nd scan */
34
35      while((p = va_arg(argp, char *)) != NULL)
36          (void)strcat(retbuf, p);
37
38      va_end(argp);
39
40      return retbuf;
41  }
```

Листинг 13 — string.c

```
1   #ifndef __SALTFS_STRING_H__
2   #define __SALTFS_STRING_H__
3
4   char *vstrcat(char const *first, ...);
5
6   #endif /*__SALTFS_STRING_H__*/
```

Листинг 14 — string.h

```
1   #include "dir.h"
2   #include "inode.h"
3   #include "user.h"
4
5   #include <linux/module.h>
6
7   #define SALTFS_MAGIC 0x5A175A17
8   #define SALTFS_BLOCKSIZE 1024
9   #define SALTFS_BLOCKSIZE_BITS 10
10  #define SALTFS_TIME_GRAN 1
11  #define SILENT_UMOUNT
12
13
14  static void salt_put_super(struct super_block *sb)
15  {
16      pr_debug("saltfs: super block destroyed\n");
17  }
18
19  static struct super_operations const salt_super_ops = {
20          .alloc_inode = salt_alloc_inode,
21          .drop_inode  = generic_delete_inode,
22          .put_super   = salt_put_super,
23  };
```

```
24
25  static int salt_fill_sb(struct super_block *sb, void *data, int silent)
26  {
27      struct inode *root = NULL;
28
29      sb->s_flags |= MS_NODIRATIME | MS_NOSUID | MS_NOEXEC;
30      sb->s_blocksize = SALTFS_BLOCKSIZE;
31      sb->s_blocksize_bits = SALTFS_BLOCKSIZE_BITS;
32      sb->s_time_gran = SALTFS_TIME_GRAN;
33      sb->s_magic = SALTFS_MAGIC;
34      sb->s_op = &salt_super_ops;
35
36      root = new_inode(sb);
37      root->i_ino = SALT_ROOT_INO;
38      root->i_sb = sb;
39      root->i_op = &simple_dir_inode_operations;
40      root->i_fop = &salt_dir_operations;
41      inode_init_owner(root, NULL, S_IFDIR | 0770);
42      update_current_time(root);
43
44      sb->s_root = d_make_root(root);
45      if (!sb->s_root) {
46          pr_err("saltfs: cannot create root\n");
47          return -ENOMEM;
48      }
49      d_set_d_op(sb->s_root, &salt_dentry_operations);
50
51      pr_debug("saltfs: inited root inode\n");
52      salt_dir_entry_create(root, "/", Salt_root, NULL);
53      pr_debug("saltfs: filled root salt_dir_entry\n");
54      salt_fill_dir(SDE(root), sb->s_root, root->i_ino, Salt_root);
55      pr_debug("saltfs: filled root directory\n");
56      pr_debug("saltfs: creating 'result' file\n");
57      salt_fill_cache_item(sb->s_root, "result", 6, Salt_result);
58
59      return 0;
60  }
61
62  static struct dentry *salt_mount(struct file_system_type *type, int flags,
63          char const *dev, void *data)
64  {
65      struct dentry *entry;
66      entry = mount_nodev(type, flags, data, salt_fill_sb);
67      if (IS_ERR(entry)) {
68          pr_err("saltfs: mounting failed\n");
69          return entry;
```

```
70      }
71      pr_info("saltfs: mounted\n");
72      return entry;
73  }
74
75  static void salt_kill_sb(struct super_block *sb)
76  {
77      salt_output_free_all();
78  #ifndef SILENT_UMOUNT
79      generic_shutdown_super(sb);
80  #endif
81      pr_info("saltfs: umounted\n");
82  }
83
84  static struct file_system_type salt_fs_type = {
85          .owner = THIS_MODULE,
86          .name = FS_NAME,
87          .mount = salt_mount,
88          .kill_sb = salt_kill_sb,
89  };
90
91  static int __init salt_init(void)
92  {
93      static unsigned long once;
94      int ret;
95
96      if (test_and_set_bit(0, &once)) {
97          pr_err("saltfs: filesystem is already mounted, "
98                  "refusing to mount it second time\n");
99          return 0;
100     }
101
102     ret = register_filesystem(&salt_fs_type);
103     if (ret != 0) {
104         pr_err("saltfs: cannot register filesystem\n");
105         return ret;
106     }
107
108     pr_debug("saltfs: filesystem registered\n");
109
110     init_proc();
111     salt_init_inodecache();
112
113     pr_debug("saltfs: module loaded\n");
114
115     return 0;
```

```
}

static void __exit salt_shutdown(void)
{
    int ret;
    pr_debug("saltfs: trying to unload module\n");
    ret = unregister_filesystem(&salt_fs_type);
    if (ret != 0)
        pr_err("saltfs: cannot unregister filesystem\n");

    pr_debug("saltfs: filesystem unregistered\n");

    shutdown_proc();

    pr_debug("saltfs: module unloaded\n");
}

module_init(salt_init);
module_exit(salt_shutdown);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Roman Inflianskas");
```

Листинг 15 — super.c

```
/* I'm using /proc/saltfs/inode + call_usermodehelper for executing
   salt and reading output.
   Linus Torvalds definitely hates me.
*/

#include "internal.h"
#include "string.h"
#include "user.h"

#include <asm/uaccess.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/slab.h>
#include <linux/string.h>

#define SALT_OUTPUT_MAX_LINE_COUNT 1024
#define SALT_OUTPUT_MAX_LINE_LENGTH 512
#define INT_MAX_STR_LENGTH 7
#define SALT_OUTPUT_PROC_ROOT "/proc/saltfs/"

```

```
22 DEFINE_SPINLOCK(salt_idr_output_lock);
23
24 struct salt_userspace_output *salt_output = NULL;
25 struct proc_dir_entry *salt_proc_root;
26 struct idr salt_output_idr;
27
28 struct salt_userspace_output *salt_output_alloc(void)
29 {
30     struct salt_userspace_output *result = (struct salt_userspace_output *)
31             kmalloc(sizeof(struct salt_userspace_output), GFP_KERNEL);
32     result->lines = (char **)
33             kcalloc(SALT_OUTPUT_MAX_LINE_COUNT, sizeof(char *), GFP_KERNEL);
34     result->line_count = 0;
35     pr_debug("saltfs: allocated salt_output\n");
36     return result;
37 }
38
39 void salt_output_free(struct salt_userspace_output *salt_output)
40 {
41     unsigned int i;
42     if (salt_output) {
43         pr_debug("saltfs: clear salt_output\n");
44         for (i = 0; i < salt_output->line_count; i++) {
45             kfree(salt_output->lines[i]);
46         }
47         kfree(salt_output->lines);
48         kfree(salt_output);
49     }
50 }
51
52 void salt_output_free_ino(int const ino)
53 {
54     salt_output_free(idr_find(&salt_output_idr, ino));
55     idr_remove(&salt_output_idr, ino);
56 }
57
58 void salt_output_free_all()
59 {
60     void *entry;
61     int ino;
62     idr_for_each_entry(&salt_output_idr, entry, ino)
63         salt_output_free_ino(ino);
64 }
65
66 static void proc_input_flush_line(char *line, unsigned int const line_length,
67         struct salt_userspace_output *salt_output)
```

```
68  {
69      if (line_length) {
70          line[line_length] = '\0';
71          salt_output->lines[salt_output->line_count] =
72                  (char *)kcalloc(line_length + 1, sizeof(char), GFP_KERNEL);
73          salt_output->lines[salt_output->line_count] =
74                  strncpy(salt_output->lines[salt_output->line_count], line,
75                          line_length);
76          pr_debug("saltfs: read '%s'\n",
77                  salt_output->lines[salt_output->line_count]);
78          salt_output->line_count++;
79      }
80  }
81
82  static ssize_t proc_write(struct file *filp, const char *buff, size_t len,
83          loff_t *off)
84  {
85      int i, ino;
86      unsigned int line_length;
87      char *line;
88      struct salt_userspace_output *salt_output;
89      sscanf(filp->f_path.dentry->d_name.name, "%d", &ino);
90      salt_output = (struct salt_userspace_output *)
91              idr_find(&salt_output_idr, ino);
92      line = (char *)
93              kcalloc(SALT_OUTPUT_MAX_LINE_LENGTH, sizeof(char), GFP_KERNEL);
94      line_length = 0;
95      pr_debug("saltfs: start reading from proc %s%d\n",
96              SALT_OUTPUT_PROC_ROOT, ino);
97      for (i = 0; line_length < SALT_OUTPUT_MAX_LINE_LENGTH - 1 && i < len;
98           i++, line_length++) {
99          get_user(line[line_length], buff + i);
100         if (line[line_length] == '\n') {
101             proc_input_flush_line(line, line_length, salt_output);
102             line_length = -1;
103         }
104     }
105     proc_input_flush_line(line, line_length, salt_output);
106     kfree(line);
107     pr_debug("saltfs: read finished\n");
108     return len;
109 }
110
111 static const struct file_operations proc_fops = {
112         .owner  = THIS_MODULE,
113         .write  = proc_write,
```

```
114         .llseek = seq_lseek,
115 };
116
117 struct salt_userspace_output *init_proc_output(int const ino,
118         char const *ino_str)
119 {
120     struct salt_userspace_output *result = salt_output_alloc();
121     idr_preload(GFP_KERNEL);
122     spin_lock(&salt_idr_output_lock);
123     idr_alloc(&salt_output_idr, result, ino, ino + 1, GFP_NOWAIT);
124     spin_unlock(&salt_idr_output_lock);
125     idr_preload_end();
126     return result;
127 }
128
129 int salt_list(char const salt_list_cmd[], int const ino)
130 {
131     int result;
132     char ino_str[INT_MAX_STR_LENGTH];
133     char *salt_list_cmd_full;
134     char *argv[] = {
135             "/usr/bin/fish",  /* Full path here      */
136             "-c",
137             "",               /* Placeholder for cmd */
138             NULL
139     };
140     static char *envp[] = {
141             "HOME=/", "TERM=linux", "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL
142     };
143     sprintf(ino_str, "%d", ino);
144     salt_list_cmd_full = vstrcat(
145             "complete --do-complete='salt_common --' >/dev/null; and ",
146             "complete --do-complete='saltfs      --' >/dev/null; and ",
147             salt_list_cmd,
148             " >" SALT_OUTPUT_PROC_ROOT, ino_str,
149             NULL);
150     argv[2] = salt_list_cmd_full;
151
152     salt_output = (struct salt_userspace_output *)
153             idr_find(&salt_output_idr, ino);
154     if (!salt_output)
155         proc_create(ino_str, 0, salt_proc_root, &proc_fops);
156     else
157         salt_output_free_ino(ino);
158     salt_output = init_proc_output(ino, ino_str);
159
```

```
    pr_debug("saltfs: executing usermodehelper; argv: '%s', '%s', '%s'\n",
            argv[0], argv[1], argv[2]);
    result = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
    pr_debug("saltfs: executed usermodehelper\n");
    kfree(salt_list_cmd_full);
    return result;
}

void init_proc(void)
{
    idr_init(&salt_output_idr);
    salt_proc_root = proc_mkdir(FS_NAME, NULL);
    pr_debug("saltfs: inited proc\n");
}

void shutdown_proc(void)
{
    remove_proc_subtree(FS_NAME, NULL);
    pr_debug("saltfs: shutdown proc\n");
}
```

Листинг 16 — user.c

```
#ifndef __SALTFS_USER_H__
#define __SALTFS_USER_H__

#include <linux/types.h>

#define SALT_LIST_CMD_FULL_MAX_LENGTH 128
#define SALT_FISH_SET_MINION(minion) \
        "set -g __fish_salt_extracted_minion ", minion, "; and "

struct salt_userspace_output {
    char **lines;
    unsigned int line_count;
};

extern struct salt_userspace_output *salt_output;
extern struct idr salt_output_idr;

int salt_list(char const salt_list_cmd[], int const ino);
void salt_output_free_all(void);
void init_proc(void);
void shutdown_proc(void);
```

```
22
23  #endif /*__SALTFS_USER_H__*/
```

Листинг 17 — user.h