|  |  |
| --- | --- |
|  | **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»

**ОТЧЕТ**

*к лабораторной работе №8*

*По курсу: «Операционные системы»*

*Тема: «Создание виртуальной файловой системы»*

Студент ИУ7-66Б

Турсунов Ж.Р.

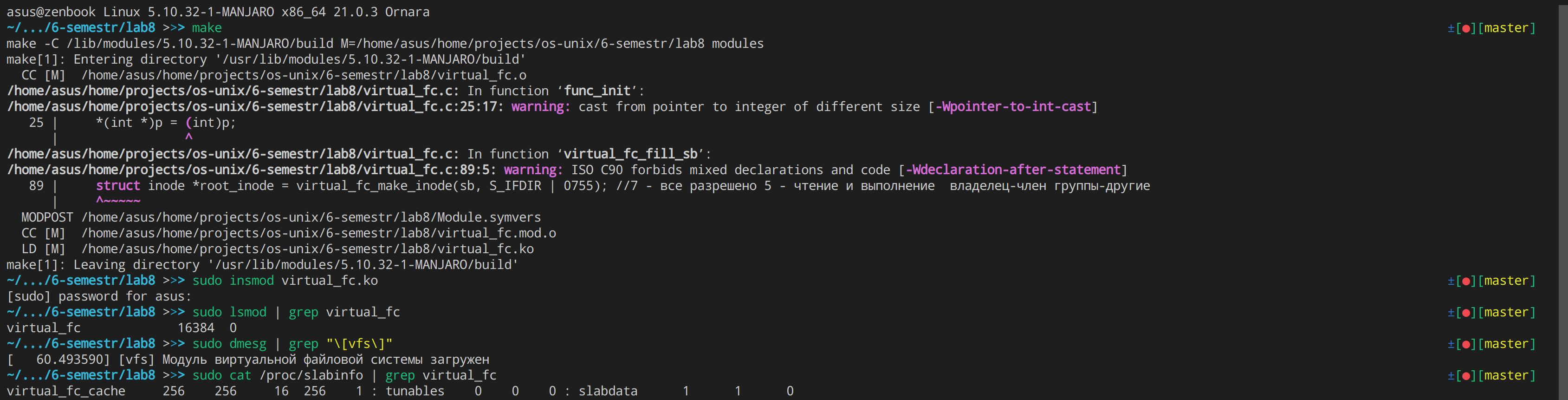
Преподаватель

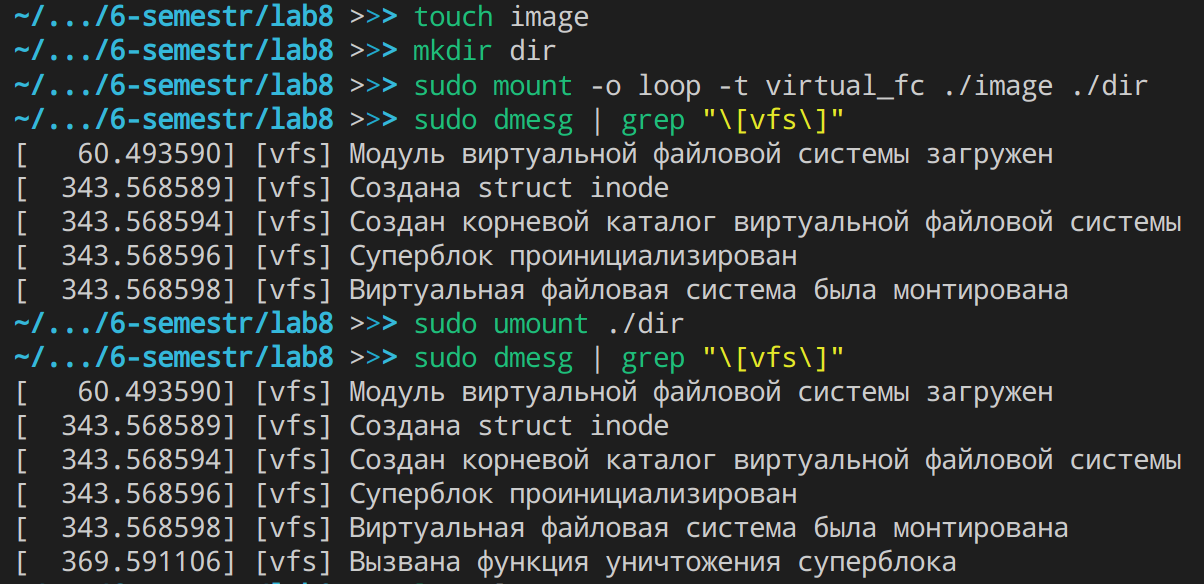
Рязанова Н.Ю.

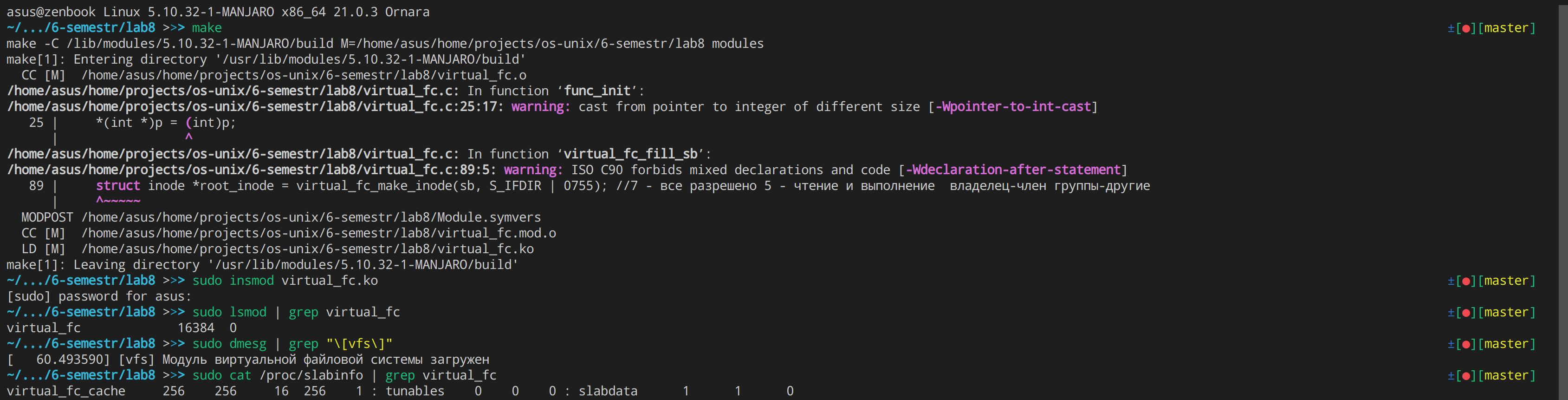
*Москва, 2021 г.*

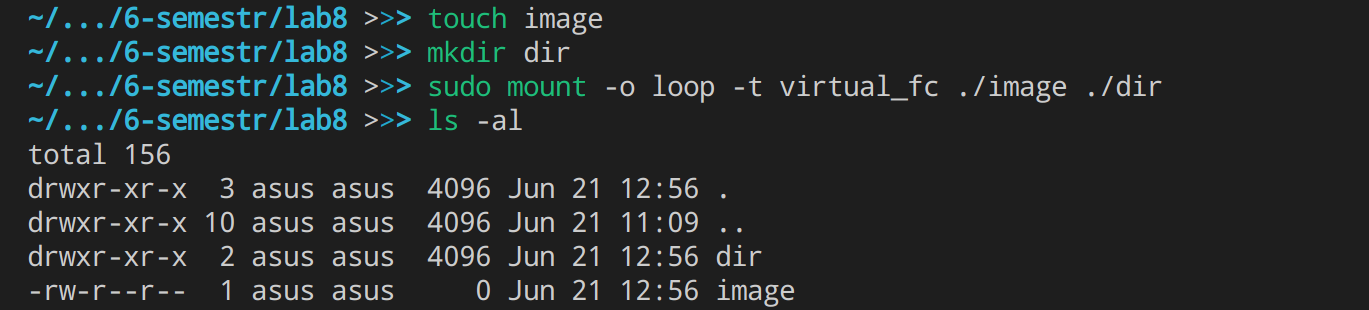
**Текст программы сервера:**

|  |
| --- |
| #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/init.h>  #include <linux/fs.h>  #include <linux/time.h>  #include <asm/atomic.h>  #include <asm/uaccess.h>  #include <linux/slab.h>  MODULE\_LICENSE**(**"GPL"**);**  MODULE\_AUTHOR**(**"Tursunov Jasurbek"**);**  #define VIRTUAL\_FC\_MAGIC\_NUM 0x13131313  #define SLAB\_NAME "virtual\_fc\_cache"  //Структура kmem\_cache содержит данные, относящиеся к конкретным CPU-модулям, набор настроек (доступных через файловую систему proc),  //статистических данных и элементов, необходимых для управления кэшем slab.  struct kmem\_cache **\***cache **=** **NULL;**  static void **\*\***cache\_mem\_area **=** **NULL;**  // Функция создания вызывается при размещении каждого элемента  static void func\_init**(**void **\***p**)**  **{**  **\*(**int **\*)**p **=** **(**int**)**p**;**  **}**  // для кеширования inode  static struct virtual\_fc\_inode  **{**  int i\_mode**;**  unsigned long i\_ino**;**  **}** virtual\_fc\_inode**;**  static struct inode **\***virtual\_fc\_make\_inode**(**struct super\_block **\***sb**,** int mode**)**  **{**  // новая структура inode  struct inode **\***ret **=** new\_inode**(**sb**);**  **if** **(**ret**)**  **{**  //директория inode NULL  inode\_init\_owner**(**ret**,** **NULL,** mode**);**  ret**->**i\_size **=** PAGE\_SIZE**;**  //current\_time(ret) размещает новую структуру inode (системным вызовом  //new\_inode()) и заполняет ее значениями: размером и временами  //(citme, atime, mtime). Повторимся, аргумент mode определяет не  //только права доступа к файлу, но и его тип - регулярный файл или каталог.  ret**->**i\_atime **=** ret**->**i\_mtime **=** ret**->**i\_ctime **=** current\_time**(**ret**);**  ret**->**i\_private **=** **&**virtual\_fc\_inode**;**  **}**  printk**(**KERN\_INFO "[vfs] Создана struct inode\n"**);**  **return** ret**;**  **}**  // вывод строки перед уничтожением суперблока  static void virtual\_fc\_put\_super**(**struct super\_block **\***sb**)**  **{**  printk**(**KERN\_INFO "[vfs] Вызвана функция уничтожения суперблока\n"**);**  **}**  // Операции структуры суперблок  static struct super\_operations const virtual\_fc\_sup\_ops **=** **{**  **.**put\_super **=** virtual\_fc\_put\_super**,**  // заглушки из libfs  **.**statfs **=** simple\_statfs**,**  **.**drop\_inode **=** generic\_delete\_inode**,**  **};**  // инициализация суперблока  static int virtual\_fc\_fill\_sb**(**struct super\_block **\***sb**,** void **\***data**,** int silent**)**  **{**  // инициализировать поля структуры суперблока  sb**->**s\_blocksize **=** PAGE\_SIZE**;** //Размер суперблока в байтах  sb**->**s\_blocksize\_bits **=** PAGE\_SHIFT**;** //Размер суперблока в битах  //магическое число, по которому драйвер файловой системы может проверить,  //что на диске хранится именно та самая файловая система, а не что-то еще  //или прочие данные  sb**->**s\_magic **=** VIRTUAL\_FC\_MAGIC\_NUM**;**  //операции суперблока  sb**->**s\_op **=** **&**virtual\_fc\_sup\_ops**;**  //построить корневой каталог ФС  // создать inode каталога ФС  // mode задает разрешения на создаваемый файл и его тип  // маска S\_IFDIR говорит функции, что мы создаем каталог  struct inode **\***root\_inode **=** virtual\_fc\_make\_inode**(**sb**,** S\_IFDIR **|** 0755**);** //7 - все разрешено 5 - чтение и выполнение владелец-член группы-другие  **if** **(!**root\_inode**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка virtual\_fc\_make\_inode\n"**);**  **return** **-**ENOMEM**;**  **}**  //Файловые и inode-операции, которые мы назначаем новому inode, взяты из libfs, т.е. предоставляются ядром.  root\_inode**->**i\_op **=** **&**simple\_dir\_inode\_operations**;**  root\_inode**->**i\_fop **=** **&**simple\_dir\_operations**;**  sb**->**s\_root **=** d\_make\_root**(**root\_inode**);**  **if** **(!**sb**->**s\_root**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка d\_make\_root\n"**);**  iput**(**root\_inode**);**  **return** **-**ENOMEM**;**  **}**  **else**  printk**(**KERN\_INFO "[vfs] Создан корневой каталог виртуальной файловой системы\n"**);**  printk**(**KERN\_INFO "[vfs] Суперблок проинициализирован"**);**  **return** 0**;**  **}**  // вызывается при монтировании ФС:  // должна вернуть структуру, описывающую корневой каталог ФС  static struct dentry **\***virtual\_fc\_mount**(**struct file\_system\_type **\***type**,** int flags**,** const char **\***dev**,** void **\***data**)**  **{**  // создается виртуальная ФС, несвязанная с устройстовом  // virtual\_fc\_fill\_sb - указатель на функцию, которая будет вызвана из mount\_nodev для создания суперблока  //nodev - Не различает файловые системы символьно-специальных и блочно-специальных устройств.  struct dentry **\***const root\_dentry **=** mount\_nodev**(**type**,** flags**,** data**,** virtual\_fc\_fill\_sb**);**  **if** **(**IS\_ERR**(**root\_dentry**))**  printk**(**KERN\_ERR "[vfs] Ошибка mount\_nodev\n"**);**  **else**  printk**(**KERN\_INFO "[vfs] Виртуальная файловая система была монтирована\n"**);**  // вернуть корневой каталог  **return** root\_dentry**;**  **}**  // Описание создаваемой ФС  static struct file\_system\_type virtual\_fc\_type **=** **{**  //поле owner отвечает за счетчик ссылок на модуль, чтобы его нельзя было случайно выгрузить.  //если файловая система была подмонтирована, то выгрузка модуля может привести к краху,  //но счетчик ссылок не позволит выгрузить модуль пока он используется, т.е. пока мы не размонтируем файловую систему.  **.**owner **=** THIS\_MODULE**,**    //поле name хранит название файловой системы. Именно это название будет использоваться при ее монтировании.  **.**name **=** "virtual\_fc"**,**  //mount и kill\_sb два поля хранящие указатели на функции.  //mount вызывается при монтировании ФС  **.**mount **=** virtual\_fc\_mount**,**  //Вызывается при размонтировании ФС  **.**kill\_sb **=** kill\_litter\_super**,**  **};**  // Инициализация модуля  static int \_\_init virtual\_fc\_init**(**void**)**  **{**  // Регистрация файловой ситемы  int ret **=** register\_filesystem**(&**virtual\_fc\_type**);**  **if** **(**ret **!=** 0**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка register\_filesystem\n"**);**  **return** ret**;**  **}**  // Выделение непр области в физической памяти  cache\_mem\_area **=** **(**void **\*\*)**kmalloc**(sizeof(**void **\*),** GFP\_KERNEL**);**  **if** **(!**cache\_mem\_area**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка cache\_mem\_area\n"**);**  kfree**(**cache\_mem\_area**);**  **return** **-**ENOMEM**;**  **}**  // создание слаб кеша  // 0 - смещение первого элемента от начала кэша (для выравнивания)  //SLAB\_HWCACHE\_ALIGN — расположение каждого элемента в слабе должно  //выравниваться по строкам процессорного кэша, это может существенно поднять производительность, но непродуктивно расходуется память;  cache **=** kmem\_cache\_create**(**SLAB\_NAME**,** **sizeof(**void **\*),** 0**,** SLAB\_HWCACHE\_ALIGN**,** func\_init**);**  **if** **(!**cache**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка kmem\_cache\_create\n"**);**  kmem\_cache\_destroy**(**cache**);**  kfree**(**cache\_mem\_area**);**  **return** **-**ENOMEM**;**  **}**  // запрос объекта  **if** **(NULL** **==** **((\***cache\_mem\_area**)** **=** kmem\_cache\_alloc**(**cache**,** GFP\_KERNEL**)))**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка kmem\_cache\_alloc\n"**);**  kmem\_cache\_free**(**cache**,** **\***cache\_mem\_area**);**  kmem\_cache\_destroy**(**cache**);**  kfree**(**cache\_mem\_area**);**  **return** **-**ENOMEM**;**  **}**  printk**(**KERN\_INFO "[vfs] Модуль виртуальной файловой системы загружен"**);**  **return** 0**;**  **}**  static void \_\_exit virtual\_fc\_exit**(**void**)**  **{**  kmem\_cache\_free**(**cache**,** **\***cache\_mem\_area**);**  kmem\_cache\_destroy**(**cache**);**  kfree**(**cache\_mem\_area**);**  **if** **(**unregister\_filesystem**(&**virtual\_fc\_type**)** **!=** 0**)**  **{**  printk**(**KERN\_ERR "[vfs] Ошибка unregister\_filesystem\n"**);**  **}**  printk**(**KERN\_INFO "[vfs] Модуль виртуальной файловой системы выгружен \n"**);**  **}**  module\_init**(**virtual\_fc\_init**);**  module\_exit**(**virtual\_fc\_exit**);** |

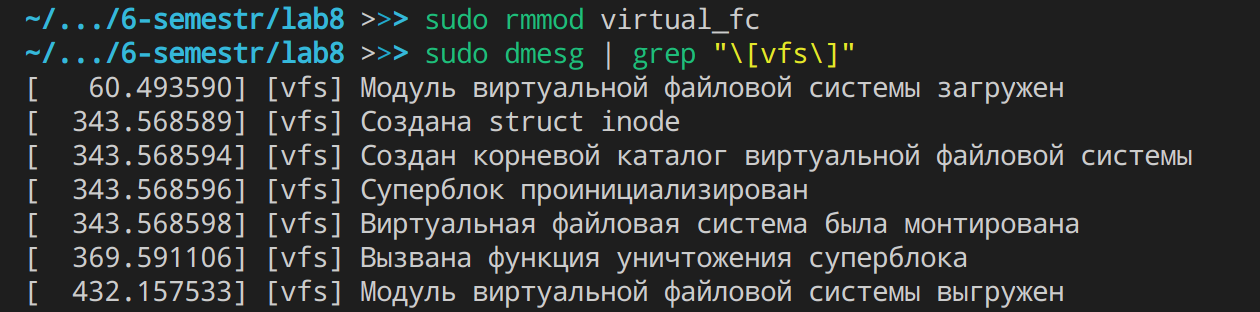
 **Рис 1: Загрузка модуля**

****

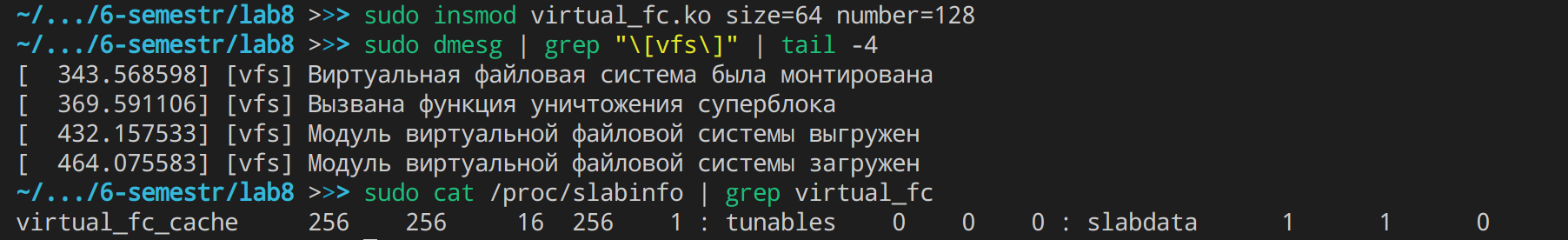
 **Рис 2: Монтирование файловой системы**

****

**Рис 3: Файловая система в дереве каталогов**



**Рис 4: Выгрузка модуля**



**Рис 5: Загрузка модуля с заданием параметров**