



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический  
университет имени Н.Э. Баумана»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

---

## Лабораторная работа № 8

Дисциплина	Операционные системы.
Тема	Создание виртуальной файловой системы.
Студент	Сиденко А.Г.
Группа	ИУ7-63Б
Оценка (баллы)	
Преподаватель	Рязанова Н.Ю.

Москва, 2020 г.

## md.c

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/pagemap.h>
5 #include <linux/fs.h>
6 #include <asm/atomic.h>
7 #include <asm/uaccess.h>
8 #include <linux/slab.h>
9
10 MODULE_LICENSE("GPL");
11 MODULE_AUTHOR("Sidenko");
12 MODULE_DESCRIPTION("Lab8");
13
14 #define VFS_MAGIC_NUMBER 0x13131313
15 #define SLABNAME "vfs_cache"
16
17 // Размер элементов кэша
18 static int size = 7;
19 // Значение параметра из командной строки, иначе значение по умолчанию
20 module_param(size, int, 0);
21 static int number = 31;
22 // Значение параметра из командной строки, иначе значение по умолчанию
23 module_param(number, int, 0);
24
25 static void* obj = NULL;
26
27 // Конструктор вызывается при размещении каждого элемента
28 static void co(void* p)
29 {
30     *(int*)p = (int)p;
31 }
32 struct kmem_cache *cache = NULL;
33
34 // Структура нужна для собственного inode
35 static struct vfs_inode
36 {
37     int i_mode;
38     unsigned long i_ino;
39 } vfs_inode;
40
41 // Создание inode
42 static struct inode * vfs_make_inode(struct super_block *sb, int mode)
43 {
44     // Размещает новую структуру inode
45     struct inode *ret = new_inode(sb);
46     if (ret)
47     {
48         // mode определяет не только права доступа, но и тип
49         inode_init_owner(ret, NULL, mode);
50     }
51 }
```

```

50 // Заполнение значениями
51 ret->i_size = PAGE_SIZE;
52 ret->i_atime = ret->i_mtime = ret->i_ctime = current_time(ret);
53 ret->i_private = &vfs_inode;
54 }
55 return ret;
56 }
57
58 // Печать строки, будет вызвана перед уничтожением суперблока
59 static void vfs_put_super(struct super_block *sb)
60 {
61     printk(KERN_DEBUG "VFS_super_block_destroyed!\n");
62 }
63
64 // Операции структуры суперблок
65 static struct super_operations const vfs_super_ops = {
66     .put_super = vfs_put_super, // деструктор суперблока
67     .statfs = simple_statfs,
68     .drop_inode = generic_delete_inode,
69 };
70
71 // Функция инициализации суперблока
72 // Создание корневого каталога ФС
73 static int vfs_fill_sb(struct super_block *sb, void *data, int silent)
74 {
75     struct inode* root = NULL;
76
77     // Заполнение структуры
78     sb->s_blocksize = PAGE_SIZE;
79     sb->s_blocksize_bits = PAGE_SHIFT;
80     sb->s_magic = VFS_MAGIC_NUMBER;
81     sb->s_op = &vfs_super_ops;
82
83     // Создание inode каталога ФС (указывает на это S_IFDIR)
84     root = vfs_make_inode(sb, S_IFDIR | 0755);
85     if (!root)
86     {
87         printk(KERN_ERR "VFS_inode_allocation_failed!\n");
88         return -ENOMEM;
89     }
90
91     // Файловые и inode-операции, которые мы назначаем новому inode
92     root->i_op = &simple_dir_inode_operations;
93     root->i_fop = &simple_dir_operations;
94
95     // Создаем dentry для представления каталога в ядре
96     sb->s_root = d_make_root(root);
97     if (!sb->s_root)
98     {
99         printk(KERN_ERR "VFS_root_creation_failed!\n");
100         iput(root);
101         return -ENOMEM;

```

```

102     }
103     return 0;
104 }
105
106 // Монтирование ФС
107 static struct dentry* vfs_mount(struct file_system_type *type, int flags,
108                                const char *dev, void *data)
109 {
110     // Так как создается виртуальная ФС
111     // несвязанная с носителем, используем
112     struct dentry* const entry=mount_nodet(type, flags, data, vfs_fill_sb);
113
114     if (IS_ERR(entry))
115         printk(KERN_ERR "VFS_mounting_failed!\n");
116     else
117         printk(KERN_DEBUG "VFS_mounted!\n");
118
119     // Корневой каталог нашей ФС
120     return entry;
121 }
122
123 // Описание создаваемой ФС
124 static struct file_system_type vfs_type = {
125     .owner = THIS_MODULE, // счетчик ссылок на модуль
126     .name = "vfs", // название ФС
127     .mount = vfs_mount, // функция, вызываемая при монтировании ФС
128     .kill_sb = kill_litter_super, // функция, вызываемая при
129                                     // размонтировании ФС
130 };
131
132 // Инициализация модуля
133 static int __init vfs_module_init(void)
134 {
135     if (size < 0)
136     {
137         printk(KERN_ERR "VFS_MODULE_invalid_sizeof_objects\n");
138         return -EINVAL;
139     }
140
141     // Выделение области, является непрерывной в физической памяти
142     obj = kmalloc(sizeof(void*), GFP_KERNEL);
143     if (!obj)
144     {
145         printk(KERN_ERR "VFS_MODULE_kmalloc_error\n");
146         return -ENOMEM;
147     }
148
149     // Создание слаба
150     // Расположение каждого элемента в слабе выравнивается по строкам
151     cache = kmem_cache_create(SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co);
152
153     if (!cache)

```

```

154 {
155     printk(KERN_ERR "VFS_MODULE_cannot_create_cache\n");
156     kfree(obj);
157     return -ENOMEM;
158 }
159
160 // Выделение объекта
161 if (NULL == (obj = kmem_cache_alloc(cache, GFP_KERNEL)))
162 {
163     printk(KERN_ERR "VFS_MODULE_cannot_alloc_object\n");
164     kmem_cache_destroy(cache);
165     return -ENOMEM;
166 }
167
168 printk(KERN_INFO "VFS_MODULE_allocate_%d_objects_into_slab:_%s\n",
169         number, SLABNAME);
170 printk(KERN_INFO "VFS_MODULE_object_size_%d_bytes, _full_size_%ld_bytes\n",
171         size, (long)size * number);
172
173 // Регистрация файловой системы
174 int ret = register_filesystem(&vfs_type);
175 if (ret != 0)
176 {
177     printk(KERN_ERR "VFS_MODULE_cannot_register_filesystem!\n");
178     kfree(obj);
179     kmem_cache_destroy(cache);
180     return ret;
181 }
182
183 printk(KERN_DEBUG "VFS_MODULE_loaded!\n");
184 return 0;
185 }
186
187 // Выход загружаемого модуля
188 static void __exit vfs_module_exit(void)
189 {
190     // Возвращение объекта
191     kmem_cache_free(cache, obj);
192
193     // Уничтожение слаба
194     kmem_cache_destroy(cache);
195     kfree(obj);
196
197     // Дeregистрация
198     if (unregister_filesystem(&vfs_type) != 0)
199     {
200         printk(KERN_ERR "VFS_MODULE_cannot_unregister_filesystem!\n");
201     }
202
203     printk(KERN_DEBUG "VFS_MODULE_unloaded!\n");
204 }
205

```

```

206 module_init(vfs_module_init);
207 module_exit(vfs_module_exit);

```

## Makefile

```

1 ifneq ($(KERNELRELEASE),)
2     obj-m := md.o
3 else
4     CURRENT = $(shell uname -r)
5     KDIR = /lib/modules/$(CURRENT)/build
6     PWD = $(shell pwd)
7
8 default:
9     $(MAKE) -C $(KDIR) M=$(PWD) modules
10
11 clean:
12     rm -rf .tmp_versions
13     rm *.ko
14     rm *.o
15     rm *.mod.c
16     rm *.symvers
17     rm *.order
18
19 endif

```

1. Загрузим модуль ядра и проверим в списке загруженных модулей ядра

```

+ [~/Desktop/lab8] $ sudo insmod md.ko
[sudo] password for parallels:
+ [~/Desktop/lab8] $ lsmod | grep md
md                16384  0

```

2. Вывод буфера сообщений ядра в стандартный поток вывода

```

+ [~/Desktop/lab8] $ sudo dmesg | tail -3
[13468.987178] VFS_MODULE allocate 31 objects into slab: vfs_cache
[13468.987179] VFS_MODULE object size 7 bytes, full size 217 bytes
[13468.987183] VFS_MODULE loaded!

```

3. Посмотрим содержимое файла /proc/slabinfo, в котором хранится информация о кэшах

```

+ [~/Desktop/lab8] $ sudo cat /proc/slabinfo | grep vfs
vfs_cache 31 240 16 240 1 : tunables 120 60 8 : slabdata 1 1 0

```

4. Создадим образ диска, пока он не хранит никаких данных. Создадим каталог, который будет точкой монтирования (корнем) файловой системы. Далее, используя этот образ, примонтируем файловую систему.

```

+ [~/Desktop/lab8] $ touch image
+ [~/Desktop/lab8] $ mkdir dir
+ [~/Desktop/lab8] $ sudo mount -o loop -t vfs ./image ./dir

```

5. Вывод буфера сообщений ядра в стандартный поток вывода

```
+ [~/Desktop/lab8] sudo dmesg | tail -4
[13468.987178] VFS_MODULE allocate 31 objects into slab: vfs_cache
[13468.987179] VFS_MODULE object size 7 bytes, full size 217 bytes
[13468.987183] VFS_MODULE loaded!
[13781.679401] VFS mounted!
```

6. Размонтируем ФС и выгрузим модуль ядра.

```
+ [~/Desktop/lab8] sudo umount ./dir
+ [~/Desktop/lab8] sudo rmmod md
```

7. Вывод буфера сообщений ядра в стандартный поток вывода

```
+ [~/Desktop/lab8] sudo dmesg | tail -6
[13468.987178] VFS_MODULE allocate 31 objects into slab: vfs_cache
[13468.987179] VFS_MODULE object size 7 bytes, full size 217 bytes
[13468.987183] VFS_MODULE loaded!
[13781.679401] VFS mounted!
[13809.682263] VFS super block destroyed!
[13820.185961] VFS_MODULE unloaded!
```

8. Также в программе предусмотрено задание размера и количества элементов кэша

```
* [~/Desktop/lab8] sudo insmod md.ko size=100 number=300
[sudo] password for parallels:
+ [~/Desktop/lab8] sudo dmesg | tail -3
[16088.546605] VFS_MODULE allocate 300 objects into slab: vfs_cache
[16088.546606] VFS_MODULE object size 100 bytes, full size 30000 bytes
[16088.546611] VFS_MODULE loaded!
+ [~/Desktop/lab8] sudo cat /proc/slabinfo | grep vfs
vfs_cache 300 320 128 32 1 : tunables 120 60 8 : slabdata 10 10 0
```