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

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/pagemap.h>
#include <linux/fs.h>
#include <asm/atomic.h>
#include <asm/uaccess.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sidenko");
MODULE_DESCRIPTION("Lab8");

#define VFS_MAGIC_NUMBER 0x13131313
#define SLABNAME "vfs_cache"

// Размер элементов кэша
static int size = 7;
// Значение параметра из командной строки, иначе значение по умолчанию
module_param(size, int, 0);
static int number = 31;
// Значение параметра из командной строки, иначе значение по умолчанию
module_param(number, int, 0);

static void* *line = NULL;

// Конструктор вызывается при размещении каждого элемента
static void co(void* p)
{
  *(int*)p = (int)p;
}
struct kmem_cache *cache = NULL;

// Структура нужна для собственного inode
static struct vfs_inode
{
  int i_mode;
  unsigned long i_ino;
} vfs_inode;

// Создание inode
static struct inode * vfs_make_inode(struct super_block *sb, int mode)
{
  // Размещает новую структуру inode
  struct inode *ret = new_inode(sb);
  if (ret)
  {
    // mode определяет не только права доступа, но и тип
    inode_init_owner(ret, NULL, mode);
```

```
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 виртуальная ФС, несвязанна с носителем, используем
111     struct dentry* const entry=mount_nodev(type, flags, data, vfs_fill_sb);
112
113     if (IS_ERR(entry))
114         printk(KERN_ERR "VFS mounting failed!\n");
115     else
116         printk(KERN_DEBUG "VFS mounted!\n");
117
118     // Корневой каталог нашей ФС
119     return entry;
120 }
121
122 // Описание создаваемой ФС
123 static struct file_system_type vfs_type = {
124     .owner = THIS_MODULE, // счетчик ссылок на модуль
125     .name = "vfs", // название ФС
126     .mount = vfs_mount, // функция, вызываемая при монтировании ФС
127     .kill_sb = kill_litter_super, // функция, вызываемая при
128                                   // размонтировании ФС
129 };
130
131 // Инициализация модуля
132 static int __init vfs_module_init(void)
133 {
134     int i;
135
136     if(size < 0)
137     {
138         printk(KERN_ERR "VFS_MODULE invalid sizeof objects\n");
139         return -EINVAL;
140     }
141
142     // Выделение области, является непрерывной в физической памяти
143     line = kmalloc(sizeof(void*) *number, GFP_KERNEL);
144     if(!line)
145     {
146         printk(KERN_ERR "VFS_MODULE kmalloc error\n");
147         kfree(line);
148         return -ENOMEM;
149     }
150
151     // Инициализация области значениями
152     for(i = 0; i < number; i++)
153         line[i] = NULL;
```

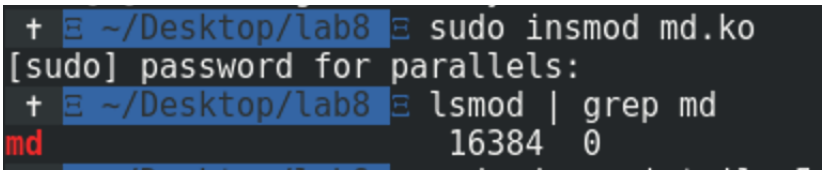
```
154
155     // Создание слаба
156     // Расположение каждого элемента в слабе выравнивается по строкам
157     cache = kmem_cache_create(SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co);
158
159     if(!cache)
160     {
161       printk(KERN_ERR "VFS_MODULE cannot create cache\n");
162       // Уничтожени слаба
163       kmem_cache_destroy(cache);
164       return -ENOMEM;
165     }
166
167     for(i = 0; i < number; i++)
168     {
169       // Выделение обтектов
170       if(NULL == (line[i] = kmem_cache_alloc(cache, GFP_KERNEL)))
171       {
172         printk(KERN_ERR "VFS_MODULE cannot alloc cache\n");
173         for(i = 0; i < number; i++ )
174           kmem_cache_free(cache, line[i]);
175         return -ENOMEM;
176       }
177     }
178
179     printk(KERN_INFO "VFS_MODULE allocate %d objects into slab: %s\n",
180                                                     number, SLABNAME);
181     printk(KERN_INFO "VFS_MODULE object size %d bytes, full size %ld bytes\n",
182                                                 size, (long)size *number);
183
184     // Регистрация файловой ситемы
185     int ret = register_filesystem(&vfs_type);
186     if (ret != 0)
187     {
188       printk(KERN_ERR "VFS_MODULE cannot register filesystem!\n");
189       return ret;
190     }
191
192     printk(KERN_DEBUG "VFS_MODULE loaded!\n");
193     return 0;
194 }
195
196 // Выход загружаемого модуля
197 static void __exit vfs_module_exit(void)
198 {
199     int i;
200     for(i = 0; i < number; i++)
201     {
202       // Возвращение обтектов
203       kmem_cache_free(cache, line[i]);
204     }
205
```

```
206     // Уничтожени слаба
207     kmem_cache_destroy(cache);
208     kfree(line);
209
210     // Дерегистрация
211     if (unregister_filesystem(&vfs_type) != 0)
212     {
213         printk(KERN_ERR "VFS_MODULE cannot unregister filesystem!\n");
214     }
215
216     printk(KERN_DEBUG "VFS_MODULE unloaded!\n");
217 }
218
219 module_init(vfs_module_init);
220 module_exit(vfs_module_exit);
```

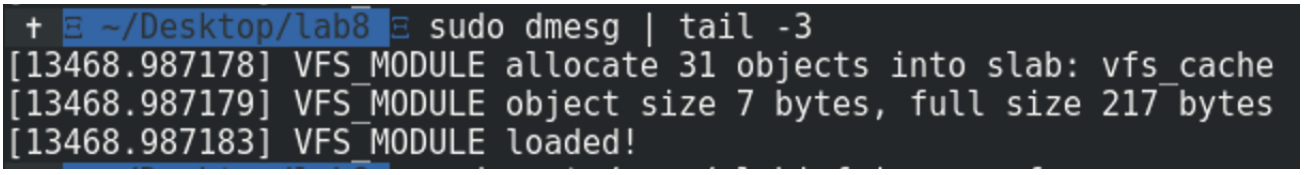
Makefile

```
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



2. Вывод буфера сообщений ядра в стандартный поток вывода



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 предумотрено задание размера и количества элементов кэша

```
 x + ~/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
```