



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
«Создание виртуальной файловой системы»*

Студент: Батбилэг Н.

Группа: ИУ7-71Б

Оценка (баллы) _____

Преподаватель: Рязанова Н.Ю.

Москва.
2021 г.

1. Задание

- Используя наработки из лабораторной работы по загружаемым модулям ядра, создать виртуальную файловую систему и slab-кэш для inode.

2. Реализация

В листинге 1 представлен код задания.

Листинг 1. Myfs.c.

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/fs.h>
4 #include <linux/time.h>
5 #include <linux/slab.h>
6 #include <linux/init.h>
7
8 MODULE_LICENSE("GPL");
9 MODULE_DESCRIPTION("MYFS module");
10 MODULE_AUTHOR("Batbileg Nomuundalai");
11
12 static int number = 0;
13 module_param(number, int, 0);
14 static struct inode **myfs_inodes = NULL;
15
16
17 #define SLABNAME "myfs_inode_cache"
18 struct kmem_cache *cache = NULL;
19
20 static const unsigned long MYFS_MAGIC_NUMBER = 0x13131313;
21
22 struct myfs_inode {
23     int i_mode;
24     unsigned long i_ino;
25 };
26
27 static int size = sizeof(struct myfs_inode);
28
29 static struct inode *myfs_make_inode(struct super_block *sb, int mode) {
30     struct inode *ret = new_inode(sb);
31     struct myfs_inode *myfs_inode = NULL;
32
33     if (ret) {
34         inode_init_owner(ret, NULL, mode);
35         ret->i_size = PAGE_SIZE;
36         ret->i_atime = ret->i_mtime = ret->i_ctime = current_time(ret);
37         myfs_inode = kmem_cache_alloc(cache, GFP_KERNEL);
38         *myfs_inode = (struct myfs_inode) {
39             .i_mode = ret->i_mode,
40             .i_ino = ret->i_ino,
41         };
42         ret->i_private = myfs_inode;
43     }
44
45     return ret;
46 }
47
48 static void myfs_put_super(struct super_block *sb) {
49     printk(KERN_DEBUG "MYFS super block destroyed!\n");
50 }
```

```

51
52 static int myfs_drop_inode(struct inode *inode) {
53     kmem_cache_free(cache, inode->i_private);
54     return generic_drop_inode(inode);
55 }
56 static struct super_operations const myfs_super_ops = {
57     .put_super = myfs_put_super,
58     .statfs = simple_statfs,
59     .drop_inode = myfs_drop_inode,
60 };
61
62 static int myfs_fill_sb(struct super_block *sb, void *data, int silent) {
63     struct inode *root = NULL;
64     int i = 0;
65
66     sb->s_blocksize = PAGE_SIZE;
67     sb->s_blocksize_bits = PAGE_SHIFT;
68     sb->s_magic = MYFS_MAGIC_NUMBER;
69     sb->s_op = &myfs_super_ops;
70
71     root = myfs_make_inode(sb, S_IFDIR | 0755);
72     if (!root) {
73         printk(KERN_ERR "MYFS make inode failed!\n");
74         return -ENOMEM;
75     }
76
77     root->i_op = &simple_dir_inode_operations;
78     root->i_fop = &simple_dir_operations;
79
80     sb->s_root = d_make_root(root);
81     if (!sb->s_root) {
82         printk(KERN_ERR "MYFS create root failed!\n");
83         iput(root);
84         return -ENOMEM;
85     }
86
87     for (i = 0; i < number; i++) {
88         myfs_inodes[i] = myfs_make_inode(sb, S_IFDIR | 0755);
89         if (!myfs_inodes[i]) {
90             printk(KERN_ERR "MYFS kmem_cache_alloc error\n");
91             for (i = 0; i < number; i++) {
92                 myfs_drop_inode(myfs_inodes[i]);
93             }
94
95             kmem_cache_destroy(cache);
96             kfree(myfs_inodes);
97             return -ENOMEM;
98         }
99     }
100     return 0;
101 }
102
103 static struct dentry *myfs_mount(struct file_system_type *type, int flags, char const *dev,
104     void *data) {
105     struct dentry *const entry = mount_nodev(type, flags, data, myfs_fill_sb);
106     if (IS_ERR(entry)) {
107         printk(KERN_ERR "MYFS mount failed!\n");
108     } else {
109         printk(KERN_DEBUG "MYFS mounted!\n");
110     }

```

```

111     return entry;
112 }
113
114 static struct file_system_type myfs_type = {
115     .owner = THIS_MODULE,
116     .name = "myfs",
117     .mount = myfs_mount,
118     .kill_sb = kill_litter_super,
119 };
120
121 static int __init myfs_init(void) {
122     int ret = register_filesystem(&myfs_type);
123     if (ret != 0) {
124         printk(KERN_ERR "MYFS_MODULE can't register FS!\n");
125         return ret;
126     }
127
128     myfs_inodes = kmalloc(sizeof(struct inode *) * number, GFP_KERNEL);
129     if (!myfs_inodes) {
130         printk(KERN_ERR "MYFS allocating error\n");
131         kfree(myfs_inodes);
132         return -ENOMEM;
133     }
134
135     cache = kmem_cache_create(SLABNAME, size, 0, SLAB_POISON, NULL);
136
137     if (!cache) {
138         printk(KERN_ERR "MYFS kmem_cache_create error\n");
139         kfree(myfs_inodes);
140         kmem_cache_destroy(cache);
141         return -ENOMEM;
142     }
143
144     printk(KERN_DEBUG "MYFS_MODULE successfully loaded\n");
145
146     return 0;
147 }
148
149 static void __exit myfs_exit(void) {
150     int i = 0;
151     int ret = unregister_filesystem(&myfs_type);
152     if (ret != 0) {
153         printk(KERN_ERR "MYFS_MODULE can't unregister FS!\n");
154     }
155     for (i = 0; i < number; i++) {
156         myfs_drop_inode(myfs_inodes[i]);
157     }
158
159     kmem_cache_destroy(cache);
160     kfree(myfs_inodes);
161     printk(KERN_DEBUG "MYFS_MODULE successfully unloaded\n");
162 }
163
164 module_init(myfs_init);
165 module_exit(myfs_exit);

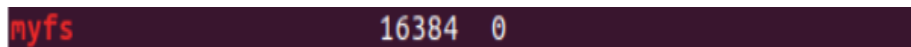
```

В листинге 2 представлено содержимое файла Makefile, необходимого для сборки программы

Листинг 2. Makefile

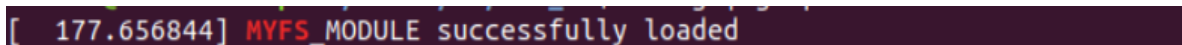
```
1 CONFIG_MODULE_SIG=n
2
3 ifneq ($(KERNELRELEASE),)
4     obj-m := myfs.o
5
6 else
7     CURRENT = $(shell uname -r)
8     KDIR = /lib/modules/$(CURRENT)/build
9     PWD = $(shell pwd)
10
11 default:
12     sudo $(MAKE) -C $(KDIR) M=$(PWD) modules
13     sudo make clean
14
15 clean:
16     rm -rf .tmp_versions
17     rm .myfs.*
18     rm *.o
19     rm *.mod.c
20     rm *.symvers
21     rm *.order
22
23 endif
```

На рисунках 1, 2, 3 представлен процесс загрузки модуля и результат. Состояние кэша: 0 активных объектов: 24 байта под каждый объект.



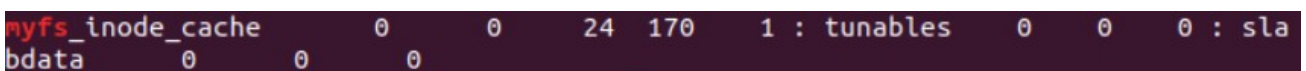
```
myfs 16384 0
```

Рисунок 1.



```
[ 177.656844] MYFS_MODULE successfully loaded
```

Рисунок 2.



```
myfs_inode_cache 0 0 0 24 170 1 : tunables 0 0 0 : sla
bdata 0 0 0
```

Рисунок 3.

На рисунках 4, 5 представлен процесс монтирования файловой системы и информация о смонтированной ФС (файловой системе). Для корневого каталога создается inode, поэтому в кэше теперь находится 1 элемент.

```
[ 177.656844] MYFS_MODULE successfully loaded
[ 351.713291] MYFS mounted!
```

Рисунок 4.

```
myfs_inode_cache      1    170    24 170    1 : tunables    0    0    0 : sla
bdata                 1    1    0
```

Рисунок 5.

На рисунках 6, 7 представлен процесс размонтирования ФС и результат. В кэше снова 0 элементов.

```
[ 177.656844] MYFS_MODULE successfully loaded
[ 351.713291] MYFS mounted!
[ 524.867918] MYFS super block destroyed!
```

Рисунок 6.

```
myfs_inode_cache      0    170    24 170    1 : tunables    0    0    0 : sla
bdata                 1    1    0
```

Рисунок 7.

На рисунке 8 представлен процесс выгрузки модуля ядра и результат.

```
[ 177.656844] MYFS_MODULE successfully loaded
[ 351.713291] MYFS mounted!
[ 524.867918] MYFS super block destroyed!
[ 570.495950] MYFS_MODULE successfully unloaded
```

Рисунок 8.

На рисунках 9, 10 представлены результаты тестирования slab-кэша при различном числе объектов. Количество выделенных объектов меняется динамически. Slab добывает страницы памяти, чтобы поместились в нем.

```
myfs_inode_cache      11    170    24 170    1 : tunables    0    0    0 : sla
bdata                 1    1    0
```

Рисунок 9.

```
myfs_inode_cache      501    510    24 170    1 : tunables    0    0    0 : sla
bdata                 3    3    0
```

Рисунок 10.