|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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»\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Лабораторная работа №** 8

Создание виртуальной файловой системы.

|  |  |
| --- | --- |
| **Дисциплина** «Операционные системы»  **Студент:** Зейналов З. Г.  **Группа:** ИУ7-61Б  **Преподаватель:** Рязанова Н. Ю. |  |

Москва.

2020 г.

**Листинг программы**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/time.h>

#include <linux/slab.h>

#define **MYFS\_MAGIC\_NUMBER** 0x13131313;

#define **SLABNAME** "my\_cache"

struct myfs\_inode

{

    int i\_mode;

    unsigned long i\_ino;

};

int inode\_number = 0;

static struct kmem\_cache \*cache;

static void **myfs\_put\_super**(struct super\_block \*sb)

{

**printk**(KERN\_DEBUG "MYFS super block destroyed\n");

}

int **free\_allocated\_inodes**(struct inode \*inode)

{

**kmem\_cache\_free**(cache, inode->i\_private);

    return 1;

}

static struct super\_operations const myfs\_super\_ops = {

    .put\_super = myfs\_put\_super,

    .statfs = simple\_statfs,

    .drop\_inode = free\_allocated\_inodes,

};

static struct inode \***myfs\_make\_inode**(struct super\_block \*sb, int mode)

{

    struct inode \*ret = **new\_inode**(sb);

    if (ret)

    {

        struct myfs\_inode \*my\_inode = **kmem\_cache\_alloc**(cache, GFP\_KERNEL);

**inode\_init\_owner**(ret, NULL, mode);

        \*my\_inode = (struct myfs\_inode){

            .i\_mode = ret->i\_mode,

            .i\_ino = ret->i\_ino

        };

        ret->i\_size = PAGE\_SIZE;

        ret->i\_atime = ret->i\_mtime = ret->i\_ctime = **current\_time**(ret);

        ret->i\_private = my\_inode;

    }

    return ret;

}

static int **myfs\_fill\_sb**(struct super\_block \*sb, void \*data, int silent)

{

    struct inode \*root = NULL;

    sb->s\_blocksize = PAGE\_SIZE;

    sb->s\_blocksize\_bits = PAGE\_SHIFT;

    sb->s\_magic = MYFS\_MAGIC\_NUMBER;

    sb->s\_op = &myfs\_super\_ops;

    root = **myfs\_make\_inode**(sb, S\_IFDIR|0755);

    if (!root)

    {

**printk**(KERN\_ERR "MYFS inode allocation failed\n");

        return -ENOMEM;

    }

    root->i\_op = &simple\_dir\_inode\_operations;

    root->i\_fop = &simple\_dir\_operations;

    sb->s\_root = **d\_make\_root**(root);

    if (!sb->s\_root)

    {

**printk**(KERN\_ERR "MYFS root creation failed\n");

**iput**(root);

        return -ENOMEM;

    }

    return 0;

}

static struct dentry\* **myfs\_mount**(struct file\_system\_type \* type, int flags, char const \*dev, void \*data)

{

    struct dentry \*const entry = **mount\_bdev**(type, flags, dev, data, myfs\_fill\_sb);

    if (**IS\_ERR**(entry))

**printk**(KERN\_ERR "MYFS mounting failed!\n");

    else

**printk**(KERN\_DEBUG "MYFS mounted");

    return entry;

}

static struct file\_system\_type myfs\_type = {

    .owner = THIS\_MODULE,

    .name = "myfs",

    .mount = myfs\_mount,

    .kill\_sb = kill\_block\_super,

};

void **co** (void \*p)

{

    \*(int \*)p = (int)p;

    inode\_number++;

}

static int \_\_init **myfs\_init**(void)

{

    int ret = **register\_filesystem**(&myfs\_type);

    cache = **kmem\_cache\_create**(SLABNAME, sizeof(struct myfs\_inode), 0, 0, co);

    if (ret != 0)

    {

**printk**(KERN\_ERR "MYFS can't register filesystem\n");

        return ret;

    }

**printk**(KERN\_INFO "MYFS filesystem registered");

    return 0;

}

static void \_\_exit **myfs\_exit**(void)

{

    int ret = **unregister\_filesystem**(&myfs\_type);

    if (ret != 0)

**printk**(KERN\_ERR "MYFS can't unregister filesystem!\n");

**kmem\_cache\_destroy**(cache);

**printk**(KERN\_INFO "MYFS unregistered %d", inode\_number);

}

**MODULE\_LICENSE**("GPL");

**MODULE\_AUTHOR**("Zeynalov Zeynal");

**module\_init**(myfs\_init);

**module\_exit**(myfs\_exit);

**Листинг Makefile’a**

ifneq ($(KERNELRELEASE),)

    obj-m   := test.o

else

    CURRENT = $(**shell** uname -r)

    KDIR = /lib/modules/$(CURRENT)/build

    PWD = $(**shell** pwd)

**default**:

    $(MAKE) -C $(KDIR) M=$(PWD) modules

**clean**:

    @rm -f \*.o .\*.cmd .\*.flags \*.mod.c \*.order

    @rm -f .\*.\*.cmd \*~ \*.\*~ TODO.\*

    @rm -fR .tmp\*

    @rm -rf .tmp\_versions

**disclean**: clean

    @rm \*.ko \*.symvers

endif

**Результаты работы программы**

Для создания виртуальной файловой системы необходимо скомпилировать загружаемый модуль ядра и загрузить его с помощью команды insmod.

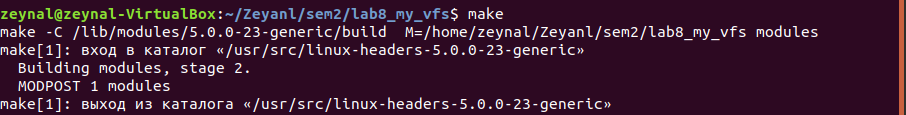


Рисунок 1 - сборка модуля ядра

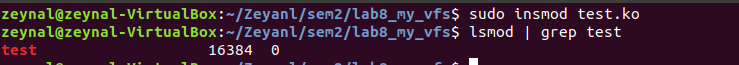


Рисунок 2 - загрузка модуля ядра.

Создадим образ диска (touch image )и корень файловой системы (mkdir dir).

Смонтируем (sudo mount -o loop -t myfs ./image ./dir) файловую систему, размонтируем (sudo umount ./dir) ее и посмотрим в syslog.

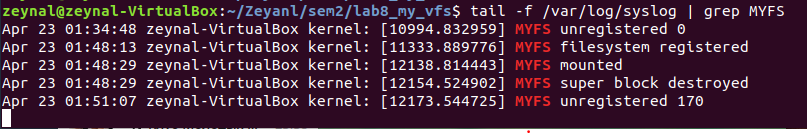


Рисунок 3 - Вывод отладочной информации в syslog

На рисунке 4 продемонстрировано состояние Slab кэша до монтирования системы.

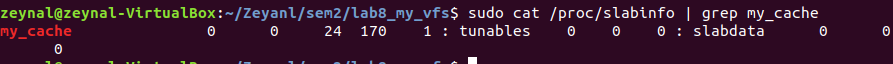


Рисунок 4 - Состояние Slab кэша до монтирования системы.

На рисунке 5 представлено состояние Slab кэша после монтирования системы.



Рисунок 5 - Состояние Slab кэша после монтирования системы.

Информация о смонтированной файловой системе представлена на рисунке 6



Рисунок 6 - Вывод информации о монтированной файловой системе

**Вывод**

В результате проделанной работы были получены навыки создания и монтирования виртуальной файловой системы. Также была изучена возможность создания slab кэша для inode.