Лабораторная работа
«Создание виртуальной файловой системы»

Примечание: при описании данной лабораторной учитывалось, что была выполнена лабораторная по загружаемым модулям ядра, и при реализации данной лабораторной необходимо использовать наработки предшествующей лабораторной работы (например, makefile).

Прежде всего необходимо подключить все необходимые библиотеки:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <inux/fs.h>
#include <linux/time.h>
```

VFS определяет интерфейс, который должны поддерживать конкретные файловые системы, чтобы работать в Linux.

Linux поддерживает большое количество файловых систем. Для того, чтобы это было возможно в VFS определена структура, определяющая тип файловой системы - struct file_system_type:

```
struct file_system_type {
        const char *name;
        int fs_flags;
#define FS_REQUIRES_DEV         1  /*требуется блочное устройство*/
#define FS_BINARY_MOUNTDATA     2 /* с версии 2.6.5 монтируемые данные являются бинарными*/
#define FS_HAS_SUBTYPE          4
#define FS_USERNS_MOUNT         8       /* Can be mounted by userns root */
#define FS_DISALLOW_NOTIFY_PERM 16      /* Disable fanotify permission events */
#define FS_RENAME_DOES_D_MOVE   32768   /* FS will handle d_move() during rename() internally. */
        int (*init_fs_context)(struct fs_context *);
        const struct fs_parameter_spec *parameters;
        struct dentry *(*mount) (struct file_system_type *, int,  const char *, void *);
        void (*kill_sb) (struct super_block *); /*прекращение доступа к суперблоку*/
        struct module *owner;           /*счетчик ссылок на файловую систему*/
        struct file_system_type * next;
        struct hlist_head fs_supers;

        struct lock_class_key s_lock_key;
        struct lock_class_key s_umount_key;
        struct lock_class_key s_vfs_rename_key;
        struct lock_class_key s_writers_key[SB_FREEZE_LEVELS];

        struct lock_class_key i_lock_key;
        struct lock_class_key i_mutex_key;
        struct lock_class_key i_mutex_dir_key;
};
```

Для каждого типа файловой системы существует только одна структура file_system_file, независимо от того, сколько таких файловых систем смонтировано и смонтирован ли хотя бы один экземпляр файловой системы.

Регистрация и монтирование файловой системы

Для связывания файловой системы с VFS необходимо определить некий минимальный набор функций и структур данных:

- struct file_system_type является глобальным "определителем" файловой системы и содержит имя ФС, а также функции инициализации и уничтожения суперблока.
- struct super_operations содержит набор функций работы с глобальными данными файловой системы. Здесь могут использоваться "заглушки", предоставляемые libfs.
- struct file_operations определяет набор функций для работы с файлами. Для файлов счетчиков мы реализуем только три из них: open, read и write; для каталогов же используем "заглушки" libfs.

Код файловой системы может быть реализован или в виде загружаемого модуля ядра, или статически связан с ядром.

Для создания новой файловой системы надо заполнить поля структуры struct file_system_file и зарегистрировать файловую систему VFS с помощью следующих функции API:

#include <linux / fs.h>
extern int register_filesystem (struct file_system_type *);
extern int unregister_filesystem (struct file_system_type *);

Переданный функци **register_filesystem() тип** struct file_system_type описывает файловую систему. Например:

```
struct file_system_file my_fs_type = {
                    . owner = THIS_MODULE,
                    . name = “my_fs”,
                    . mount = my_mount,
                   . kill_sb = my_kill_super_block,
                   . fs_flags = FS_REQUIRED_DEV,
                  };
```

Регистрация файловой системы выполняется в функции инициализации модуля. Для дерегистрации файловой системы используется функция unregister_filesystem(), которая вызывается в функции выхода загружаемого модуля.

Обе функции принимают как параметр указатель на структуру file_system_type, которая "описывает" создаваемую файловую систему. Среди всех полей этой структуры нас интересуют лишь некоторые из них, поэтому при определении структуры инициализируем только некоторые поля. Например:

```
static struct file_system_type myfs_type  =  {
        .owner  =  THIS_MODULE,
        .name  =  "myfs",
        .mount  =  myfs_mount,
        .kill_sb  =  kill_block_super,
};
```

Отсутствует поле

. fs_flags = FS_REQUIRED_DEV, но для виртуальной файловой системы можно, например, включить флаг **FS_USERNS_MOUNT**.

Где поле owner отвечает за счетчик ссылок на модуль, чтобы его нельзя было случайно выгрузить. Например, 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 файловая система не будет размонтирована.

Поле name хранит название файловой системы. Именно это название будет использоваться при ее монтировании.

mount и kill_sb два поля хранящие указатели на функции. Первая функция будет вызвана при монтировании файловой системы, а вторая при размонтировании. Достаточно реализовать всего одну, а вместо второй будем использовать kill_block_super, которую предоставляет ядро, или соответствующую generic-функцию.

Когда делается запрос на монтирование файловой системы в каталог в определенном пространстве имен, VFS вызывает соответствующий метод mount() для конкретной файловой системы. Определены следующие варианты функций mount:

```
#ifdef CONFIG_BLOCK
extern struct dentry *mount_bdev(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data,
        int (*fill_super)(struct super_block *, void *, int));
#else
static inline struct dentry *mount_bdev(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data,
        int (*fill_super)(struct super_block *, void *, int))
{
```

```
        return ERR_PTR(-ENODEV);
}
#endif
extern struct dentry *mount_single(struct file_system_type *fs_type,
        int flags, void *data,
        int (*fill_super)(struct super_block *, void *, int));
extern struct dentry *mount_nodev(struct file_system_type *fs_type,
        int flags, void *data,
        int (*fill_super)(struct super_block *, void *, int));
```

Теперь рассмотрим функцию myfs_mount. Она должна примонтировать устройство и вернуть структуру, описывающую корневой каталог файловой системы:

```
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!\n") ;
        return entry;
}
```

Замечание. Обратите внимание на обоснованность использования функции mount_bdev().

По факту, большая часть работы происходит внутри функции mount_bdev(), но нас интересует лишь ее параметр - функция myfs_fill_sb().будет вызвана из mount_bdev, чтобы проинициализировать суперблок. Сама функция myfs_mount должна вернуть структуру dentry (переменная entry), представляющую корневой каталог разрабатываемой файловой системы. Его создает именно функция myfs_fill_sb().

Функция myfs_fill_sb:

```
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;
}
```

 первую очередь заполняется структура super_block: 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; операции для суперблока, его размер. Для магического числа можно использовать любое сочетание цифр, например, значение 0x13131313.

Проинициализировав суперблок, функция myfs_fill_sb() создает корневой каталог файловой системы. Для него создается inode вызовом myfs_make_inode, реализация которого будет показана ниже. Он нуждается в указателе на суперблок и аргументе mode, который задает разрешения на создаваемый файл и его тип (маска S_IFDIR говорит функции, что мы создаем каталог). Файловые и inode-операции, которые мы назначаем новому inode, взяты из libfs, т.е. предоставляются ядром.

```
/*
 *          fs/libfs.c
 *          Library for filesystems writers.
 */
…
const struct file_operations simple_dir_operations = {
```

```
        .open           = dcache_dir_open,
        .release        = dcache_dir_close,
        .llseek         = dcache_dir_lseek,
        .read           = generic_read_dir,
        .iterate        = dcache_readdir,
        .fsync          = noop_fsync,
};
EXPORT_SYMBOL(simple_dir_operations);

const struct inode_operations simple_dir_inode_operations = {
        .lookup = simple_lookup,
};
EXPORT_SYMBOL(simple_dir_inode_operations);

static const struct super_operations simple_super_operations = {
        .statfs = simple_statfs,
};
…
```

```
static const struct super_operations simple_super_operations = {
        .statfs         = simple_statfs,
};
```

```
/*
 * the inodes created here are not hashed. If you use iunique to generate
 * unique inode values later for this filesystem, then you must take care
 * to pass it an appropriate max_reserved value to avoid collisions.
 */
int simple_fill_super(struct super_block *s, unsigned long magic,
                      struct tree_descr *files)
{
        struct inode *inode;
        struct dentry *root;
        struct dentry *dentry;
        int i;

        s->s_blocksize = PAGE_CACHE_SIZE;
        s->s_blocksize_bits = PAGE_CACHE_SHIFT;
        s->s_magic = magic;
        s->s_op = &simple_super_operations;
        s->s_time_gran = 1;

        inode = new_inode(s);
        if (!inode)
                return -ENOMEM;
        /*
         * because the root inode is 1, the files array must not contain an
         * entry at index 1
         */
        inode->i_ino = 1;
        inode->i_mode = S_IFDIR | 0755;
        inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
        inode->i_op = &simple_dir_inode_operations;
        inode->i_fop = &simple_dir_operations;
        set_nlink(inode, 2);
        root = d_make_root(inode);
        if (!root)
                return -ENOMEM;
        for (i = 0; !files->name || files->name[0]; i++, files++) {
                if (!files->name)
                        continue;

                /* warn if it tries to conflict with the root inode */
                if (unlikely(i == 1))
                        printk(KERN_WARNING "%s: %s passed in a files array"
                                "with an index of 1!\n", __func__,
                                s->s_type->name);

                dentry = d_alloc_name(root, files->name);
                if (!dentry)
```

```
                        goto out;
                inode = new_inode(s);
                if (!inode) {
                        dput(dentry);
                        goto out;
                }
                inode->i_mode = S_IFREG | files->mode;
                inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
                inode->i_fop = files->ops;
                inode->i_ino = i;
                d_add(dentry, inode);
        }
        s->s_root = root;
        return 0;
out:
        d_genocide(root);
        shrink_dcache_parent(root);
        dput(root);
        return -ENOMEM;
}
EXPORT_SYMBOL(simple_fill_super);
```

Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш. Заметим, что суперблок имеет специальное поле, хранящее указатель на dentry корневого каталога, которое также устанавливается myfs_fill_sb.

Необходимо определить структуру super_block и ее операции. В работе следует определить только одно ее поле - функцию put_super(). В put_super() определяется деструктор суперблока разрабатываемой файловой системы. Остальные поля заполняются заглушками из libfs:

```
static void myfs_put_super(struct super_block * sb)
{
        printk(KERN_DEBUG "MYFS super block destroyed!\n" ) ;
}

static struct super_operations const myfs_super_ops = {
          .put_super = myfs_put_super,
          .statfs = simple_statfs,
          .drop_inode = generic_delete_inode,
};
```

Функция myfs_put_super(_ не делает ничего полезного, она используется исключительно, чтобы напечатать в системный лог одну строчку. Функция myfs_put_super() будет вызвана внутри kill_block_super (см. выше) перед уничтожением структуры super_block, т.е. при размонтировании файловой системы.

Теперь посмотрим, как работает myfs_make_inode:

```
static struct inode * myfs_make_inode(struct super_block * sb, int mode)
{
        struct inode * ret = new_ inode(sb);

        if (ret)
        {
                inode_init_owner(ret, NULL, mode);
                ret->i_size = PAGE_SIZE;
                ret->i_atime = ret->i_mtime = ret->i_ctime = current_time(ret);
                ret->i_privat = myfs_inode;
        }
        return ret;
}
До использования myfs_inode она должна быть объявлена. Эта структура нужна для
собственного inode:
struct myfs_inode
{
    Int i_mode;
    unsigned long i_ino;
};
Это нужно сделать для кэширования inode (см. дополнение).
```

Примечание: В оригинальном источнике вместо current_time(ret) используется макрос CURRENT_TIME. Не на всех системах этот макрос работает корректно, так что в данном примере используется именно функция current_time().

Она просто 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.

Остается только написать код для инициализации модуля и его выгрузки:

```
static int __init myfs_init(void)
{
        Int ret = register_filesystem(& myfs_type);
        if(ret != 0)
```

```
        {
                printk(KERN_ERR "MYFS_MODULE cannot register filesystem!\n");
                return ret;
        }
        printk(KERN_DEBUG "MYFS_MODULE loaded !\n");
        return 0;
}


static void __exit myfs_exit(void)
{
        int ret = unregister_filesystem(&myfs_type);
        if (ret != 0)
                printk(KERN_ERR "MYFS_MODULE cannot unregister
                    filesystem !\n");


        printk(KERN_DEBUG "MYFS_MODULE unloaded !\n");
}
```

Сборка и загрузка драйвера системы ничем не отличается от сборки и загрузки обычного модуля, т.е. используются уже знакомые команды insmod и rmmod.

Вместо реального диска для экспериментов будем использовать loop устройство. Это такой драйвер "диска", который пишет данные не на физическое устройство, а в файл (образ диска). Создадим образ диска, пока он не хранит никаких данных, поэтому все просто:

touch image

Кроме того, нужно создать каталог, который будет точкой монтирования (корнем) файловой системы:

mkdir dir

Теперь, используя этот образ, примонтируем файловую систему:

sudo mount -o loop -t myfs ./image ./dir

Если операция завершилась удачно, то в системном логе можно увидеть сообщения от модуля (dmesg). Чтобы размонтировать файловую систему делаем так:

sudo umount ./dir

И опять проверяем системный лог.

Ссылки на источники:

1. Облегченная версия: *https://habrahabr.ru/company/spbau/blog/218833/*;
2. Усложненная версия: *http://opennet.ru/base/dev/virtual_fs.txt*;

Дополнение.

В лабораторной работе предлагается дополнительно создать слаб кэш для inode. Как было показано в примере, для этого нужно объявить структуру и вызвать соответствующие функции. В частности для создания кэша inode в функции myfs_init должна быть вызвана функция kmem_cache_create() (см. ниже).

Динамическое управление памятью

Цель управления памятью состоит в предоставлении метода, при помощи которого память может динамически распределяться между различными пользователями и различными целями. Метод управления памятью должен делать следующее:

- Минимизировать время, необходимое на управление памятью
- Максимально увеличить доступную память для общего использования (минимизировать ее непроизводительные расходы на управление)

Управление памятью -- в конечном счете игра с нулевым исходом в ходе поиска компромисса. Вы можете разработать алгоритм, управление которым не требует большого количества памяти, но он отнимет много времени на управление доступной памятью. Вы также можете разработать алгоритм, который эффективно управляет памятью, но использует несколько больше памяти. И наконец, требования для конкретного приложения приводят к разумному компромиссу.

Мы рассмотрим механизм, предоставляемый ядром Linux для управления памятью **slab allocation**.

Кэш slab

Распределитель памяти slab, используемый в Linux, базируется на алгоритме, впервые введенном Джефом Бонвиком (Jeff Bonwick) для операционной системы SunOS. Распределитель Джефа строится вокруг объекта кэширования. Внутри ядра значительное количество памяти выделяется на ограниченный набор объектов, например, дескрипторы файлов и другие общие структурные элементы. Джеф основывался на том, что количество времени, необходимое для инициализации регулярного объекта в ядре, превышает количество

времени, необходимое для его выделения и освобождения. Его идея состояла в том, что вместо того, чтобы возвращать освободившуюся память в общий фонд, оставлять эту память в проинициализированном состоянии для использования в тех же целях. Например, если память выделена для mutex, функцию инициализации mutex (mutex_init) необходимо выполнить только один раз, когда память впервые выделяется для mutex. Последующие распределения памяти не требуют выполнения инициализации, поскольку она уже имеет нужный статус от предыдущего освобождения и обращения к деконструктору.

Функции API

Рассмотрим API (application program interface - прикладной программный интерфейс) для создания новых кэшей slab, добавления памяти в кэш, удаления кэша, а также функции для выделения и освобождения объектов из них.

Первый этап - создание структуры кэша slab, которую можно создать статически как:

```
struct struct kmem_cache *my_cachep;
```

Эта ссылка затем используется другими функциями кэша slab для создания, удаления, распределения и т.д. 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.

Функция ядра kmem_cache_create() используется для создания нового кэша. Обычно это происходит во время инициализации ядра или при первой загрузке модуля ядра. Его прототип определен как:

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size, size_t align,
                unsigned long flags, void (*ctor)(void*, struct kmem_cache *, unsigned long),
                void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

name — строка имени кэша;
size — размер элементов кэша (единый и общий для всех элементов);
offset — смещение первого элемента от начала кэша (для обеспечения соответствующего выравнивания по границам страниц, достаточно указать 0, что означает выравнивание по умолчанию);
flags — опциональные параметры (может быть 0);
ctor, dtor — **конструктор** и **деструктор**, соответственно, вызываются при размещении-освобождении каждого элемента, но с некоторыми ограничениями ... например, деструктор будет вызываться (финализация), но не гарантируется, что это будет поисходить сразу непосредственно после удаления объекта.

К версии 2.6.24 [5, 6] он становится другим (деструктор исчезает из описания):

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size,
                size_t offset, unsigned long flags,
                void (*ctor)( void*, kmem_cache_t*, unsigned long flags ) );
```

Наконец, в 2.6.32, 2.6.35 и 2.6.35 можем наблюдать следующую фазу изменений (меняется прототип конструктора):

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size,
                size_t offset, unsigned long flags,
                void (*ctor)( void* ) );
```

Это значит, что то, что компилировалось для одного ядра, перестанет компилироваться для следующего. Вообще то, это достаточно обычная практика для ядра, но к этому нужно быть готовым, а при использовании таких достаточно глубинных механизмов, руководствоваться не навыками, а изучением заголовочных файлов текущего ядра.

Из флагов создания, поскольку они также находятся в постоянном изменении, и большая часть из них относится к отладочным опциям, стоит назвать:

SLAB_HWCACHE_ALIGN — расположение каждого элемента в слабе должно выравниваться по строкам процессорного кэша, это может существенно поднять производительность, но непродуктивно расходуется память;
SLAB_POISON — начально заполняет слаб предопределённым значением (A5A5A5A5) для обнаружения выборки неинициализированных значений;

Если не нужны какие-то особые изыски, то нулевое значение будет вполне уместно для параметра flags.

После того как кэш создан, ссылка на него возвращается функцией kmem_cache_create. Обратите внимание, что эта функция не выделяет память кешу. Вместо этого при попытке выделить объекты из кэша (изначально он пуст) ему выделяется память при помощи команды **refill**. Это та же команда, которая используется для добавления кэшу памяти, когда все его объекты израсходованы.

Как для любой операции выделения, ей сопутствует обратная операция по уничтожению слаба:

```
int kmem_cache_destroy( kmem_cache_t *cache );
```

Операция уничтожения может быть успешна (здесь достаточно редкий случай, когда функция уничтожения возвращает значение результата), только если уже **все** объекты, полученные из кэша, были возвращены в него. Таким образом, модуль должен проверить статус, возвращённый kmem_cache_destroy(); ошибка указывает на какой-то вид утечки памяти в модуле (так как некоторые объекты не были возвращены).

После того, как кэш объектов создан, вы можете выделять объекты из него, вызывая функцию kmem_cache_alloc(). Вызывающий код передает кэш, из которого выделяется объект, и набор флагов:

```
void kmem_cache_free( kmem_cache_t *cache, const void *obj );
```

Несмотря на изменчивость API слаб алокатора, вы можете охватить даже диапазон версий ядра, пользуясь директивами условной трансляции препроцессора; модуль использующий такой алокатор может выглядеть подобно следующему (архив slab.tgz):

slab.c :

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/version.h>

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "5.2" );

static int size = 7;  // для наглядности - простые числа
module_param( size, int, 0 );
static int number = 31;
module_param( number, int, 0 );
static void* *line = NULL;
static int sco = 0;
static
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,31)
void co( void* p ) {
#else
void co( void* p, kmem_cache_t* c, unsigned long f ) {
#endif
  *(int*)p = (int)p;
  sco++;
}
#define SLABNAME "my_cache"
struct kmem_cache *cache = NULL;
static int __init init( void ) {
  int i;
  if( size < sizeof( void* ) ) {
    printk( KERN_ERR "invalid argument\n" );
    return -EINVAL;
  }
  line = kmalloc( sizeof(void*) * number, GFP_KERNEL );
  if( !line ) {
    printk( KERN_ERR "kmalloc error\n" );
    goto mout;
```

```
  }
  for( i = 0; i < number; i++ )
    line[ i ] = NULL;
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,32)
  cache = kmem_cache_create( SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co, NULL );
#else
  cache = kmem_cache_create( SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co );
#endif
  if( !cache ) {
    printk( KERN_ERR "kmem_cache_create error\n" );
    goto cout;
  }
  for( i = 0; i < number; i++ )
    if( NULL == ( line[ i ] = kmem_cache_alloc( cache, GFP_KERNEL ) ) ) {
      printk( KERN_ERR "kmem_cache_alloc error\n" );
      goto oout;
    }
  printk( KERN_INFO "allocate %d objects into slab: %s\n", number, SLABNAME );
  printk( KERN_INFO "object size %d bytes, full size %ld bytes\n", size, (long)size * number );
  printk( KERN_INFO "constructor called %d times\n", sco );
  return 0;
oout:
  for( i = 0; i < number; i++ )
    kmem_cache_free( cache, line[ i ] );
cout:
  kmem_cache_destroy( cache );
mout:
  kfree( line );
  return -ENOMEM;
}
module_init( init );

static void __exit exit( void ) {
  int i;
  for( i = 0; i < number; i++ )
    kmem_cache_free( cache, line[ i ] );
  kmem_cache_destroy( cache );
  kfree( line );
}
module_exit( exit );
```

А вот как выглядит выполнение этого размещения (картина весьма поучительная, поэтому остановимся на ней подробнее):

```
$ sudo insmod ./slab.ko
$ dmesg | tail -n300 | grep -v audit
 allocate 31 objects into slab: my_cache
 object size 7 bytes, full size 217 bytes
 constructor called 257 times
$ cat /proc/slabinfo | grep my_
 # name   <active_objs> <num_objs> <objsize> ...
 my_cache      256     256      16   256     1 : tunables     0     0     0 :
slabdata      1      1      0
$ sudo rmmod slab
```

Итого: объекты размером 7 байт благополучно разместились в новом слабе с именем my_cache, отображаемом в /proc/slabinfo, организованным с размером элементов 16 байт (эффект выравнивания?), конструктор при размещении 31 таких объектов вызывался 257 раз. Обратим внимание на чрезвычайно важное обстоятельство: при создании слаба никаким образом не указывается реальный или максимальный объём памяти, находящейся под управлением этого слаба: это динамическая структура, «добирающая» столько страниц памяти, сколько нужно для поддержания размещения требуемого числа элементов данных (с учётом их размера). Увеличенное число вызовов конструктора можно отнести: а). на необходимость переразмещения существующих элементов при последующих запросах, б). эффекты SMP (2 ядра) и перераспределения данных между процессорами.

Используемые источники

1. Цирюлик О.И. Модули ядра Linux. Внутренние механизмы. http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-05.html
2. Джонс М. Анатомия распределителя памяти slab в Linux. Узнайте, как Linux управляет памятью https://www.ibm.com/developerworks/ru/library/l-linux-slab-allocator/index.html