1. Билет №12

Создание виртуальных файловых систем. Структура, описывающая файловую систему. Регистрация и дерегистрация файловой системе. Монтирование файловой системы. Точка монтирования. Кэширование в системе. Кэши SLAB, функции для работы с кэшем SLAB. Примеры из лабораторной работы. Функции, определенные на файлах (struct file_operations), функции, определенные на файлах, и их регистрация. Пример из лабораторной работы по файловой системе /proc.

1.1. Файловая подсистема

Файл — важнейшее понятие в файловой подсистеме. Файл — информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения отдельного задания или преодоления ограничений, связанных в объемом основного ЗУ.

Файл — поименованная совокупность данных, хранимая во вторичной памяти (возможно даже целая). Файл — каждая индивидуально идентифицированная единица информации.

Существует 2 ипостаси файла:

1. файл, который лежит на диске;
2. открытый файл (с которым работает процесс).

Открытый файл — файл, который открывает процесс.

Файл != место на диске. В мире современной вычислительной техники файлы имеют настолько большие размеры, что не могут храниться в непрерывном физическом адресном пространстве, они хранятся вразброс (несвязанное распределение).

Файл может занимать разные блоки/сектора/дорожки на диске аналогично тому, как память поделена на страницы. В любой фрейм может быть загружена новая страница, как и файл.

Также, важно понимать адресацию.

Соответственно, система должна обеспечить адресацию каждого такого участка.

> ОС является загружаемой программой, её не называют файлом, но когда компьютер включается, ОС находится во вторичной памяти. Затем с помощью нескольких команд, которые находятся в ПЗУ, ОС (программа) загружается в ОЗУ. При этом выполняется огромное количество действий, связанных с управлением памятью, и без ФС это сделать невозможно. Любая ОС без ФС не может быть полноценной.

Задача ФС — обеспечивать сохранение данных и доступ к сохраненным данным (обеспечивать работу с файлами).

Чтобы обеспечить хранение файла и последующий доступ к нему, файл должен быть изолирован, то есть занимать некоторое адресное пространство, и это адресное пространство должно быть защищено. Доступ обеспечивается по тому, как файл идентифицируется в системе (доступ осуществляется по его имени).

ФС — порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях информации.

> File management (управление файлами) — программные процессы, связанные с общим управлением файлами, то есть с размещением во вторичной памяти, контролем доступа к файлам, записью резервных копий, ведением справочников (directory).
>
> Основные функции управления файлами обычно возлагаются на ОС, а дополнительные — на системы управления файлами.
>
> Доступ к файлам: open, read, write, rename, delete, remove.
>
> Разработка UNIX началась с ФС. Без ФС невозможно создание приложений, работающих в режиме пользователя (сложно разделить user mode и kernel mode).
>
> Файловая подсистема взаимодействует практически со всеми модулями ОС, предоставляя пользователю возможность долговременного хранения данных, а также ОС возможность работать с объектами ядра.

1.2. Особенности файловой подсистемы Unix/Linux

В Unix все файл, если что-то не файл, то это процесс.

В системе имеются спец. файлы, про которые говорят, что они больше чем файл: программмные каналы, сокеты, внешние устройства.

Файловая система работает с регулярными (обычными) файлами и директориями. При этом Unix/Linux не делают различий между файлами и директориями.

Директория – файл, который содержит имена других файлов.

7 типов файлов в Unix:

1. '-' – обычный файл
2. 'd' – directory
3. 'l' – soft link
4. 'c' – special character device
5. 'b' – block device
6. 's' – socket
7. 'p' – named pipe

1.3. struct file_system_type

struct file_system_type определена для описания ф.с., это тип ф.с., которая будет монтироваться(команда mount).

Можно создать собственый тип ф.с.

```
1  struct file_system_type {
2    const char *name;
3    int fs_flags;
4    #define FS_REQUIRES_DEV     1
5    ...
6    #define FS_USERNS_MOUNT     8  /* Can be mounted by userns root */
7    ...
8    struct dentry *(*mount) (struct file_system_type *, int,
9    const char *, void *);
10   void (*kill_sb) (struct super_block *);
11   struct file_system_type * next;
```

```
12      struct hlist_head fs_supers;
13
14      struct lock_class_key s_lock_key;
15      struct lock_class_key s_umount_key;
16      struct lock_class_key s_vfs_rename_key;
17      ...
18    };
```

1.4. Создание собственной файловой системы

Чтобы создать собственую ф.с. В struct superblock есть поле file_system_type (структура ядра)

После описания ф.с., ядро предоставляет возможность зарегистрировать/удалить ф.с.

Структура описывающая конкретный тип ф.с. может быть только 1. При этом одна и та же ф.с. мб подмонтирована много раз.

Пример создания собств. ф.с.

Инициализация полей структуры file_system_type

```
1   struct file_system_type fs_type =
2   {
3       .owner = THIS_MODULE,
4       .name = "myfs",
5       .mount = myfs_mount,
6       .kill_sb = kill_litter_super
7   }
```

В функции myfs_mount можно вызвать mount_bdev/ mount_nodef/ mount_single

При создании ФС мы инициализируем лишь следующие поля:

— owner - нужно для организации счетчика ссылок на модуль (нужен, чтобы система не была выгружена, когда фс примонтирована).

— name - имя ФС.

— mount - указатель на функцию, которая будет вызвана при монтировании ФС.

— kill_sb - указатель на функцию, которая будет вызвана при размонтировании ФС.

Разработчик ф.с. должен определить набор функций для работы с файлами в своей ф.с. Для этого используется struct file_operations.

1.5. Регистрация и дерегистрация файловой системы

Для регистрации ф.с. ядро предоставляет ф-цию register_filesystem()(для удаления unregister_filesystem). Функции register_filesystem передается инициализированная структура file_system_type.

1.6. Монтирование файловой системы. Точка монтирования

Фактически VFS — интерфейс, с помощью которого ОС может работать с большим количеством файловых систем.

Основной такой работы (базовым действием) является монтирование: прежде чем файловая система станет доступна (мы сможем увидеть ее каталоги и файлы) она должна быть смонтирована.

Монтирование — подготовка раздела диска к использованию файловой системы. Для этого в начале раздела диска выделяется структура super_block, одним из полей которой является список inode, с помощью которого можно получить доступ к любому файлу файловой системы.

Когда файловая система монтируется, заполняются поля struct vfsmount, которая представляет конкретный экземпляр файловой системы, или, иными словами, точку монтирования. Точкой монтирования является директория дерева каталогов.

Вся файловая система должна занимать либо диск, либо раздел диска и начинаться с корневого каталога.

Любая файловая система монтируется к общему дереву каталогов (монтируется в поддиректорию).

И эта подмонтированная файловая система описывается суперблоком и должна занимать некоторый раздел жесткого диска ("это делается в процессе монтирования").

Когда файловая система монтируется, заполняются поля структуры super_block.

super_block содержит информацию, необходимую для монтирования и управления файловой системой.

> Пример: мы хотим посмотреть содержимое флешки. Флешка имеет свою файловую систему, она может быть подмонтирована к дереву каталогов, и ее директории, поддиректории и файлы, которые мы сохраним на флешке, будут доступны. Потом мы достаем флешку. "Хорошая"система контролирует это и сделает демонтирование файловой системы за нас.

Если в системе присутствует некоторый образ диска image, а также создан каталог, который будет являться точкой монтирования файловой системы dir, то подмонтировать файловую систему можно, используя команду: mount -o loop -t myfs ./image ./dir

Параметр -o указывает список параметров, разделенных запятыми. Одним из прогрессивных типов монтирования, является монтирование через петлевое (loop, по сути, это «псевдоустройство» (то есть устройство, которое физически не существует — виртуальное блочное устройство), которое позволяет обрабатывать файл как блочное устройство) устройство. Если петлевое устройство явно не указано в строке (а как раз параметр -o loop это задает), тогда mount попытается найти неиспользуемое в настоящий момент петлевое устройство и применить его.

Аргумент следующий за -t указывает тип файловой системы.

./image - это устройство. ./dir - это каталог.

umount — команда для размонтирования файловой системы:

umount ./dir

1.7. Кэширование в системе

1.7.1. Кеш inode

Задача кеш inode — ускорение поиска и доступа.
Кеш inode в Linux:

1. Глобальный хеш-массив inode_hash_table

   В нем каждый inode хешируется по значению указателя на superblock и 32-разрядному номеру inode. Если superblock отсутствует, то inode добавляется к двусвязному списку anon_hash_chain. Такие inode называют *анонимными*. Например сокеты, которые создаются вызовом ф-ции sock_alloc, которая вызывает get_empty_inode()

2. Глобальный список inode_in_use содержит допустимые inode, у которых i_count $>$ 0, i_nlink $>$ 0. Только что созданные inode добавляются в этот список.

3. Глобальный список inode_unused. В нем находятся допустимые inode с i_count=0

4. Для каждого superblock, который содержит inode с i_count > 0, i_nlink > 0 и i_state – dirty создается список этих inode. inode отмечается как грязный, когда он был изменен. Он добавлется в список f_dirty, но только если inode был хеширован

5. SLAB cache называется inode_cacher

1.8. Кэши SLAB

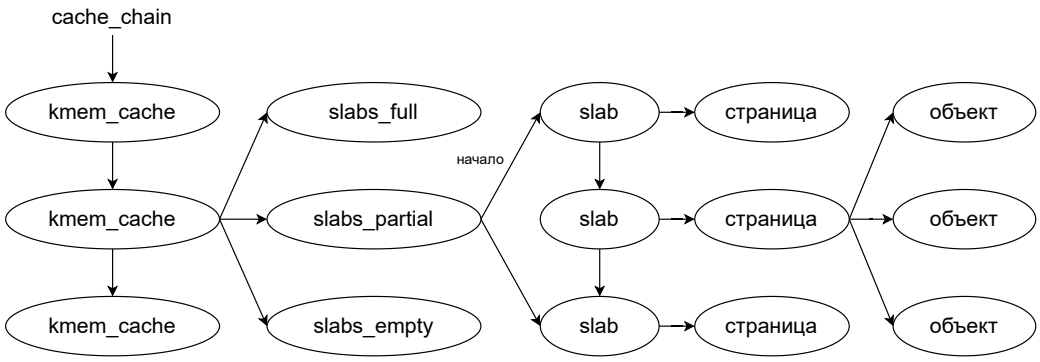
Данный подход управления памятью обеспечивает устранение фрагментации памяти -> управление памятью выполняется эффективно.

Смысл такого выделения памяти: кол-во типов, котрыми оперирует разработчик, невелико.

Объект – экземпляр структуры, созданный для файла, дтиректории и т.д.

Загрузка и выгрузка объектов приводит к фрагментации. В рез-те наблюдений было установлено, что часто используются одни и те же объекты. Поэтому нет смысла омвобождать выделенный учсток памяти, т.к. этот же участок памяти можно будет еще раз выделить. Т.е. после удаления в программе проинициализированного объекта память не освобождается, а записывается в соотв. SLAB кеш

ПРи следующем создании объекта такого же типа выделение происходит на основе SLAB cache. SLAB представляет собой непрерывный участок памяти (обычно неск. смежных стр.) и может состоять из одного или более слабов.



Каждый кеш содержит список слабов

Существует 3 слаба:

1. slabs_full – заполненный

2. slabs_partial – частично заполненный

3. slabs_empty – пустой

Объекты – основные элементы, которые выделяются из спец. кеша и в него же возвращаются

slabs_empty – основные кандидаты на повторное использование

В случае распределения SLAB участки памяти, подходящие под размещение объектов данных определенного типа, определены заранее. Аллокатор SLAB (распределитель) хранит информацию о размещении этих участков, к-ые также известны как кеш

В результате, если поступает зпрос на выделение памяти для объекта определенного типа (скорее именно размера), то он удовлетворяется с помощью SLAB

1.9. Функции для работы с кэшем SLAB. Примеры из лабораторной работы

```
1  #define SLAB_NAME "my_vfs_cache"
2
3  static struct kmem_cache *cache = NULL;
4  static void **cache_mem_area = NULL;
5
6  static struct file_system_type my_vfs_type = {
7     .owner = THIS_MODULE,
8     .name = "myvfs",
9     .mount = my_vfs_mount,
10    .kill_sb = my_kill_super,
11 };
12
13 static void func_init(void *p)
14 {
15    *(int *)p = (int)p;
16 }
17
18 static int __init my_vfs_init(void){
19    int rc = register_filesystem(&my_vfs_type);
20    // error handling
21    if ((cache_mem_area = kmalloc(sizeof(void*), GFP_KERNEL)) == NULL)
22    // error handling
23    if ((cache = kmem_cache_create(SLAB_NAME, sizeof(my_vfs_inode), 0,
          SLAB_HWCACHE_ALIGN, func_init)) == NULL)
24    // error handling
```

```
25      if (((*cache_mem_area) = kmem_cache_alloc(cache, GFP_KERNEL)) == NULL)
26      // ...
27      return 0;
28    }
29
30    static void __exit my_vfs_exit(void)
31    {
32      kmem_cache_free(cache, *cache_mem_area);
33      kmem_cache_destroy(cache);
34      kfree(cache_mem_area);
35      int rc = unregister_filesystem(&my_vfs_type);
36      // ...
37    }
38
39    module_init(my_vfs_init);
40    module_exit(my_vfs_exit);
```

1.10. Функции, определенные на файлах (struct file_operations), функции, определенные на файлах, и их регистрация

1.10.1. Определение struct file_operations

```
1   struct file_operations {
2     struct module *owner;
3     loff_t (*llseek) (struct file *, loff_t, int);
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
          *);
6     ...
7     int (*open) (struct inode *, struct file *);
8     ...
9     int (*release) (struct inode *, struct file *);
10    ...
11  };
```

1.10.2. Регистрация функций для работы с файлами

Разработчики драйверов должны регистрировать свои функции read/write.

В Unix/linux все файл, чтобы все действия свести к однотипным операциям (read/write) и не "размножать" эти действия, а свести к небольшому набору операций.

Для регистрации своих функций read/write в драйверах используется struct file_operations

С некоторой версии ядра появилась struct proc_ops. В загружаемых модулях ядра можно использовать условную компиляцию

```
1  #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,6,0)
2  #define HAVE_PROC_OPS
3  #endif
4  #ifdef HAVE_PROC_OPS
5  static struct proc_ops fops = {
6      .proc_read = fortune_read,
7      .proc_write = fortune_write,
8      .proc_open = fortune_open,
9      .proc_release = fortune_release,
10 };
11 #else
12 static struct file_operations fops = {
13     .owner = THIS_MODULE,
14     .read = fortune_read,
15     .write = fortune_write,
16     .open = fortune_open,
17     .release = fortune_release,
18 };
19 #endif
```

proc_open и open имеют одни и те же формальные параметры (указатели на struct inode и на struct file)

С остальными функциями аналогично. struct proc_ops сделана, чтобы не вешаться на функции struct file_operations, которые используются драйверами. Функции struct file_operations настолько важны для работы системы, что их решили освободить от работы с ф.с. proc

1.11. Пример из лабораторной работы по файловой системе /proc

```
1  #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,16,0)
2  #define HAVE_PROC_OPS
3  #endif
4
5  #define MAX_COOKIE_BUF_SIZE PAGE_SIZE
6
7  static ssize_t fortune_write(struct file *file, const char __user *buf,
       size_t len, loff_t *ppos)
8  {
9    // ...
10   if (copy_from_user(&cookie_buffer[write_index], buf, len) != 0)
11   // error handling
12   write_index += len;
13   cookie_buffer[write_index - 1] = '\0';
14   return len;
15 }
16
17 static ssize_t fortune_read(struct file *file, char __user *buf, size_t
       len, loff_t *f_pos)
18 {
19   // ...
20   int read_len = snprintf(tmp_buffer, MAX_COOKIE_BUF_SIZE, "%s\n", &
       cookie_buffer[read_index]);
21   if (copy_to_user(buf, tmp_buffer, read_len) != 0)
22   // error handling
23   read_index += read_len;
24   *f_pos += read_len;
25   return read_len;
26 }
27
28 #ifdef HAVE_PROC_OPS
29 static struct proc_ops fops = {
30   .proc_read = fortune_read,
31   .proc_write = fortune_write,
32   .proc_open = fortune_open,
33   .proc_release = fortune_release,
34 };
35 #else
```

```
36  static struct file_operations fops = {
37      .owner = THIS_MODULE,
38      .read = fortune_read,
39      .write = fortune_write,
40      .open = fortune_open,
41      .release = fortune_release,
42  };
43  #endif
44
45  static int __init fortune_init(void)
46  {
47      if ((cookie_buffer = vzalloc(MAX_COOKIE_BUF_SIZE)) == NULL)
48      // error handling
49      if ((fortune_dir = proc_mkdir(FORTUNE_DIRNAME, NULL)) == NULL)
50      // error handling
51      if ((fortune_file = proc_create(FORTUNE_FILENAME, S_IRUGO | S_IWUGO,
            fortune_dir, &fops)) == NULL)
52      // error handling
53      if ((fortune_symlink = proc_symlink(FORTUNE_SYMLINK, NULL,
            FORTUNE_PATH)) == NULL)
54      // error handling
55      printk(KERN_INFO "␣+␣module␣is␣loaded.\n");
56      return 0;
57  }
58
59  static void __exit fortune_exit(void)
60  {
61      // cleanup
62      printk(KERN_INFO "␣+␣module␣is␣unloaded.\n");
63  }
64
65  module_init(fortune_init);
66  module_exit(fortune_exit);
```