Лекция 2.1. Блокирующие и неблокирующие операции

Разработал: Максимов А.Н.

Версия 1.5. 01.2024

Содержание

- Альтернативный способ создания специальных файлов символьных
- Блокирующее и неблокирующее поведение

Зарезервированные старшие номера

Официальный реестр всех доступных устройств для ОС Linux. Формально он называется LANANA – Linux Assigned Names And Numbers Authority! Только эти люди могут официально присвоить устройствам узел устройства – тип и major:minor номера

```
1 char        Memory devices
                1 = /dev/mem          Physical memory access
                2 = /dev/kmem         OBSOLETE - replaced by /proc/kcore
                3 = /dev/null         Null device
                4 = /dev/port         I/O port access
                5 = /dev/zero         Null byte source
                6 = /dev/core         OBSOLETE - replaced by /proc/kcore
                7 = /dev/full         Returns ENOSPC on write
                8 = /dev/random       Nondeterministic random number gen.
                9 = /dev/urandom      Faster, less secure random number gen.
               10 = /dev/aio          Asynchronous I/O notification interface
               11 = /dev/kmsg         Writes to this come out as printk's, reads
                                      export the buffered printk records.
               12 = /dev/oldmem       OBSOLETE - replaced by /proc/vmcore
```

https://www.kernel.org/doc/Documentation/admin-guide/devices.txt

Еще один способ регистрации символьного драйвера

Существует возможность нескольким драйверам разделять старший номер устройства.

int res = register_chrdev_region(dev_t first, uint count, char *dev_name);

- dev_t first первый номер устройства(major / minor) для региона
- uint count общее количество устройств (может major-number границу)
- char *dev_name device name (имя устройства появляется в /proc/devices)

Параметр &fops не передается.

Автоматическое создание и удаление специального файла устройства

Создание

```
alloc_chrdev_region();

cdev_init();
cdev_add();

class_create();
device_create();
```

Удаление

```
unregister_chrdev_region();

cdev_del();

class_destroy();
device_destroy();
```

Динамическое выделение номера устройства

```
int register_chrdev_region(dev_t first, uint count, char *dev_name);
```

Динамическое выделение номера устройства

```
int alloc_chrdev_region( dev_t *dev,uint firstminor, uint count, char
    *dev_name);
```

```
dev_t dev;
res = alloc_chrdev_region(&dev, 0, 1, dev_name);
if (res<0) return res;
   major = MAJOR(dev);
```

Регистрация. Удаление.

```
int register_chrdev_region(dev_t first, uint
  count, char*dev_name);
```

или

```
int alloc_chrdev_region( dev_t *dev, uint
  firstminor, uint count, char *dev_name);
```

Освобождения региона номеров (в the cleanup function)

```
void unregister_chrdev_region(dev_t first,
  uint count);
```

Пример

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

dev_t dev = 0;
static int __init foo_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "foo_dev")) <0){
                pr_err("Cannot allocate major number for device\n");
                return -1;
        }
        pr_info("Kernel Module Inserted Successfully...\n");
        return 0;
}
static void __exit foo_exit(void)
{
        unregister_chrdev_region(dev, 1);
        pr_info("Kernel Module Removed Successfully...\n");
}

module_init(foo_init);
module_exit(foo_exit);
MODULE_LICENSE("GPL");
```

Добавление структуры cdev

Структура cdev - это внутренняя структура ядра, представляющая символьные устройства.

structure cdev (<linux/cdev.h>) представляет устройство.

Добавление структуры cdev в ядро:

int cdev_add(struct cdev *,dev_t num, uint count);

Удаление из ядра (в cleanup):

void cdev_del(struct cdev *);

Структура cdev.Связь с fops

У нас нет &fops в **`register_chrdev_region`** для связи со структурой file-operations structure – используется другой способ.

Структура cdev представляет символьное устройства (<linux/cdev.h>).

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

или

```
struct cdev my_cdev;
cdev_init(&my_cdev, &my_fops);
my_cdev->owner = THIS_MODULE;
```

https://embetronicx.com/tutorials/linux/device-drivers/cdev-structure-and-file-operations-of-character-drivers/

Итого init_module

```
int foo_init(void) {
  int major, res=0;
  res = alloc_chrdev_region(&dev, 0, 1, dev_name);
  if (res<0) { return res; }
  major = MAJOR(dev);
  my_cdev = cdev_alloc();
  my_cdev->ops   = &fops;
  my_cdev->owner = THIS_MODULE;
  res = cdev_add(my_cdev, dev, 1);
  if (res<0){
    unregister_chrdev_region(dev, 1);
    return res;
  }
  printk(" %s init - major: %d  \r\n", dev_name, major);
  return 0;
}
```

Итого cleanup_module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>                 // struct file_operations
#include <linux/cdev.h>
#include <asm/uaccess.h>      // put_user
#include <linux/tty.h>                // tty
MODULE_LICENSE("GPL");
static char dev_name[]="rwk2_dev";              // Имя в /proc/devices
static char me[64]; static char *mess="The goal of this tutorial ";
static char *mp; static int  debu = 2;
module_param(debu, int,   0);

void foo_exit(void){
   cdev_del(my_cdev);
  unregister_chrdev_region(dev, 1);
  printfk(" %s remove \r\n", dev_name);
}
```

Как создать файл устройства автоматически?

Демон udev можно с помощью его конфигурационных файлов настроить дополнительно и точно указать имена файлов устройств, права доступа к ним, их типы и т. д. Так что касается драйвера, требуется с помощью API моделей устройств Linux, объявленных в <linux/device.h>, заполнить в /sys соответствующие записи. Все остальное делается с помощью udev. Класс устройства создается следующим образом:

struct class *cl = **class_create**(THIS_MODULE, "<device class name>");

Затем в этот класс информация об устройстве (<major, minor>) заносится следующим образом:

device_create(cl, NULL, first, NULL, "<device name format>", ...);

Здесь, в качестве first указывается dev_t. Соответственно, дополняющими или обратными вызовами, которые должны вызыватся в хронологически обратном порядке, являются:

device_destroy(cl, first);

class_destroy(cl);

http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-05.html

Как создать файл устройства автоматически?

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/err.h>
#include <linux/device.h>
dev_t dev = 0;
static struct class *dev_class;
static int __init foo_init(void) {
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){                    //Allocating Major number
                return -1;  // Cannot allocate major number for device
        }
        pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
        dev_class = class_create(THIS_MODULE,"etx_class");                      //Allocating Major number
        if(IS_ERR(dev_class)){
            goto r_class; // Cannot create the struct class for device
        }
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){    // Creating device
            goto r_device; // Cannot create the Device
        }
        return 0;  // Kernel Module Inserted Successfully
r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}
```

Как создать файл устройства автоматически?

```
static void __exit hello_world_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        unregister_chrdev_region(dev, 1);
        pr_info("Kernel Module Removed Successfully...\n");
}
```

device_create — создать устройство и зарегистрировать в sysfs

```
struct device * device_create (    struct class * class,
      struct device * parent,dev_t devt,void * drvdata,
      const char * fmt,...);
```

Аргументы:

class - pointer to the struct class that this device should be registered to

Parent - pointer to the parent struct device of this new device, if any

devt - the dev_t for the char device to be added

Drvdata - the data to be added to the device for callbacks

Fmt - string for the device's name

... variable arguments

Практический пример.

.

Реализовать драйвер с инициализацией и деинициализацеией в новом стиле и автоматическим созданием файла устройства.

Еще раз про реализацию read

Вызов open() в программе пользовательского уровня – это системный вызов. Он вызывает функцию ядра. Эта функция инициализирует необходимые данные. После этого вызывается метод open драйвера.

Прототип функции read

ssize_t (*read) (struct file *filp,char *buffer, size_t len, loff_t *offs);

the following parameters are provided by the caller (kernel) :

struct file *filp – укасзатель на file structure

char *buffer – указательна буфер в пространстве плользователя

size_t len – число байт, которые надо прочесть

loff_t *offs – указатель на поле f_pos в структуре file

Возвращаемое значение:

- положительно знечение – число байт успешно прочитанных (может быть меньше len – это не ошибка)

- 0 - end of file (не ошибка)

(если еще остались данные для чтение устройство должно блокироваться)

- отрицательное значение - error (см. <asm/errno.h>)

Реализация read. Обмен с user space

static ssize_t device_read (struct file *filp, char *buffer, size_t len, loff_t *offs);

В практическом примере реализуем read без аппаратного обеспечения- будет читать из внутреннего буфера. Будет передавать по 10 байт за вызов read

Обмен данными с user_space:

#include <asm/uaccess.h>

put_user (char kernel_item, char *user_buff);

Или

```
ulong copy_to_user(void *to, void *from, ulong bytes);
```

Причины – свопирование памяти, обеспечение безопасности.

Wolfgang Koch. Linux Device Drivers. Lecture 3.

Реализация read. Обмен с user space

Т.к. память в пространстве пользователя может быть выгружена на диск используются специальные функции, которые могут обрабатывать такую ситуацию. Ошибка промаха страницы может привести к засыпанию процесса. Наш метод должен быть реентерабельныцм и работать в нескольких контекстах – статусная информация не должна хранится в глобальных пемеременных.

Есть макросы и функции которые не производят проверки и потому более быстрые:

_ _put_user(item, ptr);

ulong _ _copy_to_user(void *to, void *from, ulong bytes);

(использует access_ok())

Реализация read. Обмен с user space

```
static char mess[]="The goal of this tutorial ";
static char *mp; // mp=mess; in init() or open()
static ssize_t device_read(struct file *filp, char *buffer, size_t len, loff_t *offs) {
unsigned int i;
for(i=0; i<10; i++){
   if(i==len) break;
   if(*mp==0) break;
   put_user(*mp++, buffer++);
}
return i;
}
```

Тестовое пользовательское приложение app1

```
int fd, k=1 ;
char inbu[100];
fd = open("/dev/mydev", O_RDONLY);
...
while(k>0){
  k = read(fd,inbu,14);
  if (k<0){ perror(" read "); break;}
   inbu[k]=0;
   printf(" read %2d : %s \n", k, inbu);
}
```

Практический пример.

Драйвер с функцией read

Пользовательское приложение app1

Практический пример.

Вызвать app1 :

read 10 : The goal o

read 10 : f this tut

read 6 : orial

read 0 :

Вызвать еще раз app1 :

??? Каков вывод

open

`open()` выполняет необходимую инициализацию для дальнейшей работы:

Увеличивает счетчик использования (не обязательно в Linux 2.4, 2.6)

_ проверяет ошибки устройства (device not ready)

_ инициализирует устройство, если оно открыто в первый раз

_ определяет младший номер устройства

_ выделяет и заполняет структуру (`filp->private_data`)

Обратные метод - `release()`

```
static int device_release(struct inode
  *inode,
struct file *filp);
```

open

Запустите приложени app1 два раза. Наш драйвер работает не как обычный файл:

> app1 > app2

read 10 : The goal o

read 10 : f this tut

read 6 : orial

read 0 :

read 0 :

В обоих случаях используется один указатель *mp,

Являющийся глобальной переменной (инициализирован в open(), используется в read())

Работа с контекстом. Структура file

Использование параметра : **`struct file *filp`**

Структура определена в (`<linux/fs.h>`) определяет открытый файл в пространстве ядра

Создается ядром в процессе системного вызова open и передается в качестве параметра.

Представляют интерес следующие поля:

`struct file_operations *f_op;` – может быть подменен для работы с несколькими различными младшими номерами устройств

`loff_t f_pos` – текущая позиция чтения или записи. Драйве не должен напрямую изменять этот указатель. Вместо этого – использовать последний агрумент read и write. `loff_t` изменяется в lseek.

unsigned int f_flags; - флаги как O_RDONLY, O_NONBLOCK, and O_SYNC.

void *private_data – динамически выделяемые данные.

Работа с контекстом. Структура file

`void *private_data;` будем использовать для сохранения контеста:

```
static int device_open(struct inode *inode,struct file *filp) {
 filp->private_data = mess;
 return 0;
}
static ssize_t device_read(struct file *filp, char *buffer, size_t
   len, loff_t *offs) {
unsigned int i; char *mp;
mp = filp->private_data;
for(i=0; i<10; i++){
if(i==len) break;
if(*mp==0) break;
put_user(*mp++, buffer++);
}
filp->private_data = mp;
return i;
}
```

Как узнать minor nymber

Есть вызов unsigned iminor(struct inode *inode); - возвращает младший номер устройства для inode

```
static int device_open(struct inode *inode,struct file *filp) {
  unsigned myminor;
  myminor = iminor(inode);
  filp->private_data = buffer_i[myminor];
}
static ssize_t device_read(struct file *filp, char *buffer, size_t
   len, loff_t *offs) {
   unsigned int i;
   char *mp;
   mp = filp->private_data;
   // iminor(filp->f_path.dentry->d_inode) - тут можно узнать так
   return i;
}
```

Практическое задание.

Запустить файл. Попробовать использование контекста.

Приме пользовательского приложения для работы с символьным драйвером

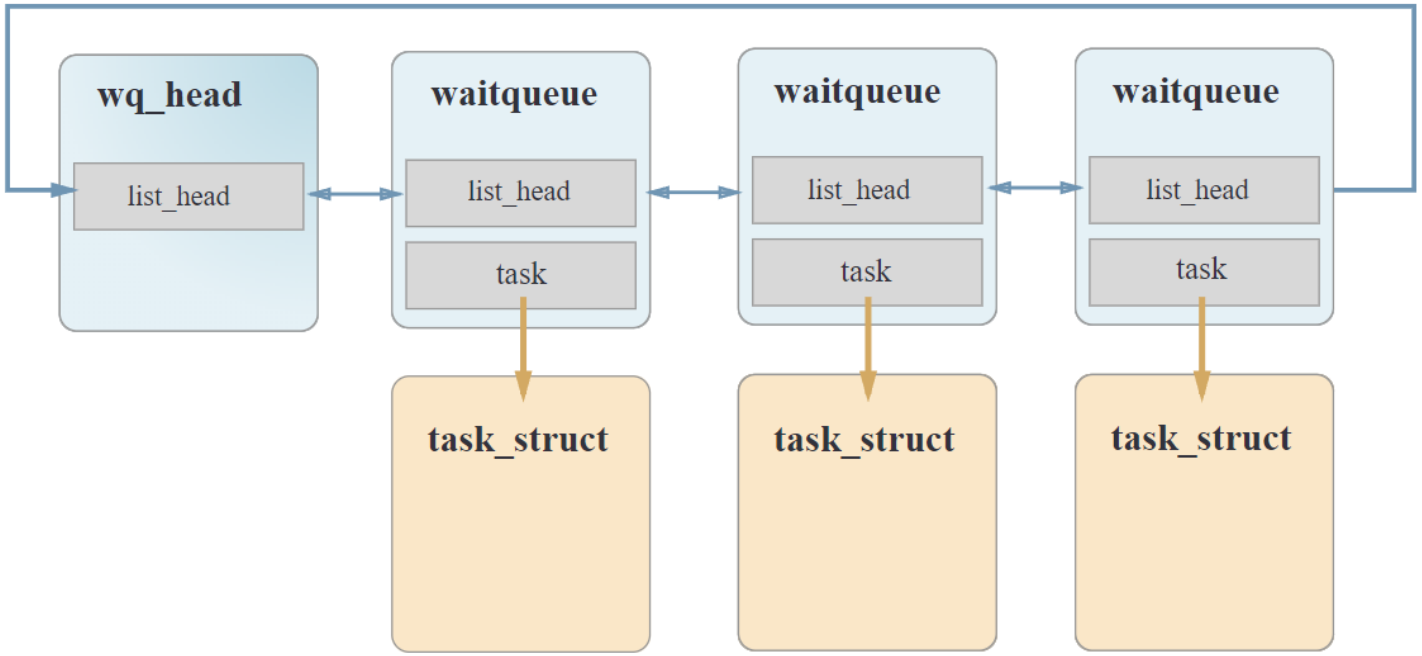
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main() {
    int fd;
    char buf[100];
    fd = open("/dev/foo",O_RDWR);
    read(fd,buf,20);
    buf[20]=0;
    printf("Input: >>> %s <<<\n",buf);
  close(fd);
}
```

Блокирующее поведение

При работе с устройством вызовы read, write, ioctl могут переводить текущую задачу в состояние TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE, если данные не готовы.

Блокировка осуществляется драйвером путем помещения задачи в очередь ожидания wait_queue



https://cs4118.github.io/www/2023-1/lect/10-run-wait-queues.html

Поместить процесс в сон

Новое определение (исключает гонки)

void wait_event(wq, condition);

int wait_event_interruptible(wq, condition);

long wait_event_timeout(wq, condition,n_jiffies);

или

wait_event_interruptible()

- возвращает -ERESTARTSYS если прервана сигналом

if (wait_event_interruptible(wq, condition))

return -ERESTARTSYS;

Пробуждение процесса

Пробуждает все процессы, которые ждут в очереди:

wake_up(wait_queue_head_t *);

wake_up_interruptible(wait_queue_head_t *);

Последняя функции пробуждает только процесс, который ждет с учетом сигналов

Можно использовать wake_up() в обоих случаях

wake_up_interruptible() часто используется в обработчиках прерываний

Логика работы

Помещение процесса в сон

wait_event_interruptible() логика работы:

– устанавливает состояние процесса в TASK_INTERRUPTIBLE

(it is in an interruptible sleep)

– задача добавляется в очередь ожидания (wait queue)

– вызов функции schedule приводит к освобождения процессора текущей задачей (context switch)

Пробуждение процесса:

schedule возвращает управление только, если кто-то вызовет функцию wake_up(), которая установит состояние процесса в TASK_RUNNING и удалит процесс из очереди ожидания(wait queue)

Пример использования блокирующих операций

```
ssize_t read(struct file *filp, char *buff, size_t count, loff_t *offp)  {
    printk(KERN_INFO "Inside read\n");
    printk(KERN_INFO "Scheduling Out\n");
    wait_event_interruptible(wq, flag == 'y');
    flag = 'n';
    printk(KERN_INFO "Woken Up\n");
    return 0;
}

ssize_t write(struct file *filp, const char *buff, size_t count, loff_t *offp)  {
    printk(KERN_INFO "Inside write\n");
    if (copy_from_user(&flag, buff, 1)) {
        return -EFAULT;
    }
    printk(KERN_INFO "%c", flag);
    wake_up_interruptible(&wq);
    return count;
}
```

https://sysplay.in/blog/linux-kernel-internals/2015/12/waiting-blocking-in-linux-driver-part-3/

Неблокирующий ввод-вывод

Пользовательское приложени при открытии указывает флаг O_NONBLOCK

```
fd=open(DEV_NAME, O_RDWR | O_NONBLOCK);
```

Драйвер может проверить filp->f_flags:

```
if ((ir == iw) // buffer empty
&& (filp->f_flags & O_NONBLOCK))
    return -EAGAIN;
if (wait_event_interruptible(wq, (ir!=iw)))
    return -ERESTARTSYS;
```

Задание

Реализовать возможность межзадачного обмена между двумя приложениями при помощи символьного драйвера

Литература

1. https://embetronicx.com/tutorials/linux/device-drivers/device-file-creation-for-character-drivers/
2. simple linux driver code to demo the blocking read and non-blocking read https://gist.github.com/itrobotics/0fe54adfff5a14e2be9f
3. Basic Character Driver in Linux https://linuxhint.com/basic-character-driver-linux/
4. https://embetronicx.com/tutorials/linux/device-drivers/cdev-structure-and-file-operations-of-character-drivers/
5. Unleashing the power of Linux: A step-by-step guide for writing a Linux device driver for an ultrasonic sensor on a Raspberry Pi - Part 1

https://chrizog.com/linux-device-driver-tutorial-ultrasonic-sensor-1

6. https://linuxhint.com/basic-character-driver-linux/