МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
имени Н.Э. БАУМАНА

Учебное пособие

Методические указания
по выполнению домашних заданий
по единому комплексному заданию по блоку дисциплины

«Системное программирование»

МГТУ имени Н.Э. Баумана

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
имени Н.Э. БАУМАНА

Методические указания
по выполнению домашних заданий
по единому комплексному заданию по блоку дисциплины

«Системное программирование»

Москва
МГТУ имени Н.Э. Баумана

2012

УДК 681.3.06(075.8)
ББК 32.973-018
И201

Методические указания по выполнению домашних заданий по единому комплексному заданию по блоку дисциплины «Системное программирование» / Коллектив авторов –
М.: МГТУ им. Н.Э. Баумана, 2012. – 12 с.: ил.

В методических указаниях рассмотрены основные этапы, их последовательность и содержание по выполнению домашних заданий курсовой работы по единому комплексному заданию по блоку дисциплины «Системное программирование».

Ил. 39. Табл. 5. Библиогр. 7 назв.

УДК 681.3.06(075.8)

Работа с файлами устройств

Файлы устройств представляют физические устройства. В большинстве своем, физические устройства используются как для вывода, так и для ввода, таким образом необходимо иметь некий механизм для передачи данных от процесса (через модуль ядра) к устройству. Один из вариантов -- открыть файл устройства и записать в него данные, точно так же, как в обычный файл. В следующем примере, операция записи реализуется функцией device_write.

Однако, этого не всегда бывает достаточно. Допустим, что у вас есть модем, подключенный к компьютеру через последовательный порт (это может быть и внутренний модем, с точки зрения CPU он "выглядит" как модем, связанный с последовательным портом). Естественное решение -- использовать файл устройства для передачи данных модему (это могут быть команды модема или данные, которые будут посланы в телефонную линию) и для чтения данных из модема (ответы модема на команды или данные, полученные из телефонной линии). Однако, это оставляет открытым вопрос о том, как взаимодействовать непосредственно с последовательным портом, например, как настроить скорость обмена.

Ответ: в Unix следует использовать специальную функцию с именем ioctl (сокращенно от Input Output ConTroL). Любое устройство может иметь свои команды ioctl, которые могут читать (для передачи данных от процесса ядру), писать (для передачи данных от ядра к процессу), и писать и читать, и ни то ни другое. Функция ioctl вызывается с тремя параметрами: дескриптор файла устройства, номер ioctl и третий параметр, который имеет тип long, используется для передачи дополнительных аргументов.

Номер ioctl содержит комбинацию бит, составляющих старший номер устройства, тип команды и тип дополнительного параметра. Обычно номер ioctl создается макроопределением (_IO, _IOR, _IOW или _IOWR, в зависимости от типа) в файле заголовка. Этот заголовочный должен подключаться директивой #include, к исходным файлам программы, которая использует ioctl для обмена данными с модулем. В примере, приводимом ниже, представлены файл заголовка chardev.h и программа, которая взаимодействует с модулем ioctl.c.

Если вы предполагаете использовать ioctl в ваших собственных модулях, то вам надлежит обратиться к файлу Documentation/ioctl-number.txt с тем, чтобы не "занять" зарегистрированные номера ioctl.

ПРИМЕР 6-1. CHARDEV.C

```
/*
 *  chardev.c - Пример создания символьного устройства
 *              доступного на запись/чтение
 */

#include <linux/module.h> /* Необходимо для любого модуля */
#include <linux/kernel.h> /* Все-таки мы работаем с ядром! */
#include <linux/fs.h>
#include <asm/uaccess.h>        /* определения функций get_user и put_user */
```

```c
#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

/*
 * Устройство уже открыто? Используется для
 * предотвращения конкурирующих запросов к устройству
 */
static int Device_Open = 0;

/*
 * Ответ устройства на запрос
 */
static char Message[BUF_LEN];

/*
 * Позиция в буфере.
 * Используется в том случае, если сообщение оказывется длиннее
 * чем размер буфера.
 */
static char *Message_Ptr;

/*
 * Вызывается когда процесс пытается открыть файл устройства
 */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk("device_open(%p)\n", file);
#endif

  /*
   * В каждый конкретный момент времени только один процесс может открыть файл
устройства
   */
  if (Device_Open)
    return -EBUSY;

  Device_Open++;
  /*
   * Инициализация сообщения
   */
  Message_Ptr = Message;
  try_module_get(THIS_MODULE);
  return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk("device_release(%p,%p)\n", inode, file);
#endif

  /*
   * Теперь мы готовы принять запрос от другого процесса
```

```
   */
  Device_Open--;

  module_put(THIS_MODULE);
  return SUCCESS;
}

/*
 * Вызывается когда процесс, открывший файл устройства
 * пытается считать из него данные.
 */
static ssize_t device_read(struct file *file, /* см. include/linux/fs.h   */
            char __user * buffer,             /* буфер для сообщения */
            size_t length,                    /* размер буфера       */
            loff_t * offset)
{
  /*
   * Количество байт, фактически записанных в буфер
   */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n", file, buffer, length);
#endif

  /*
   * Если достигнут конец сообщения -- вернуть 0
   * (признак конца файла)
   */
  if (*Message_Ptr == 0)
    return 0;

  /*
   * Собственно запись данных в буфер
   */
  while (length && *Message_Ptr) {

    /*
     * Поскольку буфер располагается в пространстве пользователя,
     * обычное присвоение не сработает. Поэтому
     * для записи данных используется put_user,
     * которая копирует данные из пространства ядра
     * в пространство пользователя.
     */
    put_user(*(Message_Ptr++), buffer++);
    length--;
    bytes_read++;
  }

#ifdef DEBUG
  printk("Read %d bytes, %d left\n", bytes_read, length);
#endif

  /*
   * Вернуть количество байт, помещенных в буфер.
   */
  return bytes_read;
```

```
}

/*
 * Вызывается при попытке записи в файл устройства
 */
static ssize_t
device_write(struct file *file,
             const char __user * buffer, size_t length, loff_t * offset)
{
  int i;

#ifdef DEBUG
  printk("device_write(%p,%s,%d)", file, buffer, length);
#endif

  for (i = 0; i < length && i < BUF_LEN; i++)
    get_user(Message[i], buffer + i);

  Message_Ptr = Message;

  /*
   * Вернуть количество принятых байт
   */
  return i;
}

/*
 * Вызывается, когда процесс пытается выполнить операцию ioctl над файлом устройства.
 * Кроме inode и структуры file функция получает два дополнительных параметра:
 * номер ioctl и дополнительные аргументы.
 *
 */
int device_ioctl(struct inode *inode, /* см. include/linux/fs.h */
      struct file *file,              /* то же самое */
      unsigned int ioctl_num,         /* номер и аргументы ioctl */
      unsigned long ioctl_param)
{
  int i;
  char *temp;
  char ch;

  /*
   * Реакция на различные команды ioctl
   */
  switch (ioctl_num) {
  case IOCTL_SET_MSG:
    /*
     * Принять указатель на сообщение (в пространстве пользователя)
     * и переписать в буфер. Адрес которого задан в дополнительно аргументе.
     */
    temp = (char *)ioctl_param;

    /*
     * Найти длину сообщения
     */
    get_user(ch, temp);
    for (i = 0; ch && i < BUF_LEN; i++, temp++)
```

```
      get_user(ch, temp);

    device_write(file, (char *)ioctl_param, i, 0);
    break;

  case IOCTL_GET_MSG:
    /*
     * Передать текущее сообщение вызывающему процессу -
     * записать по указанному адресу.
     */
    i = device_read(file, (char *)ioctl_param, 99, 0);

    /*
     * Вставить в буфер завершающий символ \0
     */
    put_user('\0', (char *)ioctl_param + i);
    break;

  case IOCTL_GET_NTH_BYTE:
    /*
     * Этот вызов является вводом (ioctl_param) и
     * выводом (возвращаемое значение функции) одновременно
     */
    return Message[ioctl_param];
    break;
  }

  return SUCCESS;
}

/* Объявлнеия */

/*
 * В этой структуре хранятся адреса функций-обработчиков
 * операций, производимых процессом над устройством.
 * Поскольку указатель на эту структуру хранится в таблице устройств,
 * она не может быть локальной для init_module.
 * Отсутствующие указатели в структуре забиваются значением NULL.
 */
struct file_operations Fops = {
  .read = device_read,
  .write = device_write,
  .ioctl = device_ioctl,
  .open = device_open,
  .release = device_release,    /* оно же close */
};

/*
 * Инициализация модуля - Регистрация символьного устройства
 */
int init_module()
{
  int ret_val;
  /*
   * Регистрация символьного устройства (по крайней мере - попытка регистрации)
   */
  ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);
```

```
  /*
   * Отрицательное значение означает ошибку
   */
  if (ret_val < 0) {
    printk("%s failed with %d\n",
           "Sorry, registering the character device ", ret_val);
    return ret_val;
  }

  printk("%s The major device number is %d.\n",
         "Registeration is a success", MAJOR_NUM);
  printk("If you want to talk to the device driver,\n");
  printk("you'll have to create a device file. \n");
  printk("We suggest you use:\n");
  printk("mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
  printk("The device file name is important, because\n");
  printk("the ioctl program assumes that's the\n");
  printk("file you'll use.\n");

  return 0;
}

/*
 * Завершение работы модуля - дерегистрация файла в /proc
 */
void cleanup_module()
{
  int ret;

  /*
   * Дерегистрация устройства
   */
  ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

  /*
   * Если обнаружена ошибка -- вывести сообщение
   */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

ПРИМЕР 6-2. CHARDEV.H

```
/*
 *  chardev.h - определения ioctl.
 *
 *  Определения, которые здесь находятся, должны помещаться в заголовочный файл
потому,
 *  что они потребуются как модулю ядра (chardev.c), так и
 *  вызывающему процессу (ioctl.c)
 */

#ifndef CHARDEV_H
#define CHARDEV_H
```

```
#include <linux/ioctl.h>

/*
 * Старший номер устройства. В случае использования ioctl,
 * мы уже лишены возможности воспользоваться динамическим номером,
 * поскольку он должен быть известен заранее.
 */
#define MAJOR_NUM 100

/*
 * Операция передачи сообщения драйверу устройства
 */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/*
 * _IOR означает, что команда передает данные
 * от пользовательского процесса к модулю ядра
 *
 * Первый аргумент, MAJOR_NUM -- старший номер устройства.
 *
 * Второй аргумент -- код команды
 * (можно указать иное значение).
 *
 * Третий аргумент -- тип данных, передаваемых в ядро
 */

/*
 * Операция получения сообщения от драйвера устройства
 */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
 * Эта команда IOCTL используется для вывода данных.
 * Нам по прежнему нужен буфер, размещенный в адресном пространстве
 * вызывающего процесса, куда это сообщение должно быть переписано.
 */

/*
 * Команда получения n-ного байта сообщения
 */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
 * Здесь команда IOCTL работает как на ввод, так и на вывод.
 * Она принимает от пользователя номер байта (n),
 * и возвращает n-ный байт сообщения (Message[n]).
 */

/*
 * Имя файла устройства
 */
#define DEVICE_FILE_NAME "char_dev"

#endif
```

ПРИМЕР 6-3. IOCTL.C

```
/*
```

```
 *  ioctl.c - Пример программы, использующей ioctl для управления модулем ядра
 *
 *  До сих пор мы ползовались командой cat, для передачи данных в/из модуля.
 *  Теперь же мы должны написать свою программу, которая использовала бы ioctl.
 */

/*
 * Определения старшего номера устройства и коды операций ioctl
 */
#include "chardev.h"

#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */

/*
 * Функции работы с драйвером через ioctl
 */

ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;

  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

  if (ret_val < 0) {
    printf("Ошибка при вызове ioctl_set_msg: %d\n", ret_val);
    exit(-1);
  }
}

ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];

  /*
   * Внимание - ядро понятия не имеет -- какой длины буфер мы используем
   * поэтому возможна ошибка, связанная с переполнением буфера.
   * В реальных проектах вам необходимо предусмотреть
   * передачу в ioctl двух дополнительных параметров:
   * собственно буфера сообщения и его длину
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

  if (ret_val < 0) {
    printf("Ошибка при вызове ioctl_get_msg: %d\n", ret_val);
    exit(-1);
  }

  printf("Получено сообщение (get_msg): %s\n", message);
}

ioctl_get_nth_byte(int file_desc)
{
  int i;
  char c;
```

```
  printf("N-ный байт в сообщении (get_nth_byte): ");

  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

    if (c < 0) {
      printf
        ("Ошибка при вызове ioctl_get_nth_byte на %d-м байте.\n", i);
      exit(-1);
    }

    putchar(c);
  }
  putchar('\n');
}

/*
 * Main - Проверка работоспособности функции ioctl
 */
main()
{
  int file_desc, ret_val;
  char *msg = "Это сообщение передается через ioctl\n";

  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf("Невозможно открыть файл устройства: %s\n", DEVICE_FILE_NAME);
    exit(-1);
  }

  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);

  close(file_desc);
}
```

ПРИМЕР 6-4. MAKEFILE

```
obj-m += chardev.o
```

Для облегчения сборки примера, предлагается скрипт, который выполнит эту работу за вас:

```
#/bin/sh

# сборка пользовательского приложения
gcc -o ioctl ioctl.c

# создание файла устройства
mknod char_dev c 100 0
```