



[\[Главная \]](#) [\[Гостевая \]](#)

[Назад](#) | [Содержание](#) | [Вперед](#)

6.9. Блокировка доступа к файлам.

В базах данных нередко встречается ситуация одновременного доступа к одним и тем же данным. Допустим, что в некотором файле хранятся данные, которые могут читаться и записываться произвольным числом процессов.

- Допустим, что процесс А изменяет некоторую область файла, в то время как процесс В пытается прочесть ту же область. Итогом такого соревнования может быть то, что процесс В прочтет неверные данные.
- Допустим, что процесс А изменяет некоторую область файла, в то время как процесс С также изменяет ту же самую область. В итоге эта область может содержать неверные данные (часть – от процесса А, часть – от С).

Ясно, что требуется механизм синхронизации процессов, позволяющий не пускать другой процесс (процессы) читать и/или записывать данные в указанной области. Механизмов синхронизации в *UNIX* существует множество: от семафоров до блокировок областей файла. О последних мы и будем тут говорить.

Прежде всего отметим, что блокировки файла носят в *UNIX* **необязательный** характер. То есть, программа не использующая вызовов синхронизации, будет иметь доступ к данным без каких либо ограничений. Увы. Таким образом, программы, собирающиеся корректно пользоваться общими данными, должны **все** использовать – и при том один и тот же механизм синхронизации: заключить между собой "джентльменское соглашение".

6.9.1. Блокировка устанавливается при помощи вызова

```
flock_t lock;  
  
fcntl(fd, operation, &lock);
```

Здесь **operation** может быть одним из трех:

F_SETLK

Устанавливает или снимает замок, описываемый структурой **lock**. Структура **flock_t** имеет такие поля:

```
short  l_type;  
short  l_whence;  
off_t  l_start;  
size_t l_len;  
  
long   l_sysid;  
pid_t  l_pid;
```

l_type

тип блокировки:

```
F_RDLCK - на чтение;  
F_WRLCK - на запись;  
F_UNLCK - снять все замки.
```

l_whence, l_start, l_len

описывают сегмент файла, на который ставится замок: от точки *lseek*(fd, **l_start**, **l_whence**); длиной **l_len** байт. Здесь **l_whence** может быть: *SEEK_SET*, *SEEK_CUR*, *SEEK_END*. **l_len** равное нулю означает "до конца файла". Так если все три параметра равны 0, то будет заблокирован весь файл.

F_SETLKW

Устанавливает или снимает замок, описываемый структурой **lock**. При этом, если замок на область, пересекающуюся с указанной уже кем-то установлен, то сперва дождаться снятия этого замка.

Пытаемся поставить замок на	Нет чужих замков	Уже есть замок на READ	уже есть замок на WRITE	
-----	-----			READ
WRITE		читать	читать	ждать;запереть;читать
UNLOCK		записать	ждать;запереть;записать	ждать;запереть;записать
		отпереть	отпереть	отпереть

- Если кто-то читает сегмент файла, то другие тоже могут его читать свободно, ибо чтение не изменяет файла.
- Если же кто-то записывает файл – то все остальные должны дожидаться окончания записи и разблокировки.
- Если кто-то читает сегмент, а другой процесс собрался изменить (записать) этот сегмент, то этот другой процесс обязан дожидаться окончания чтения первым.
- В момент, обозначенный как **отпереть** – будятся процессы, ждущие разблокировки, и ровно один из них получает доступ (может установить свою блокировку). Порядок кто из них будет первым – вообще говоря не определен.

F_GETLK

Запрашиваем возможность установить замок, описанный в **lock**.

- Если мы можем установить такой замок (не заперто никем), то в структуре **lock** поле **l_type** становится равным **F_UNLCK** и поле **l_whence** равным **SEEK_SET**.
- Если замок уже кем-то установлен (и вызов **F_SETLKW** заблокировал бы наш процесс, привел бы к ожиданию), мы получаем информацию о чужом замке в структуру **lock**. При этом в поле **l_pid** заносится идентификатор процесса, создавшего этот замок, а в поле **l_sysid** – идентификатор машины (поскольку блокировка файлов поддерживается через сетевые файловые системы). Замки автоматически снимаются при закрытии дескриптора файла. Замки *не* наследуются порожденным процессом при вызове **fork**.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>

char DataFile [] = "data.xxx";
char info [] = "abcdefghijklmnopqrstuvwxy";
#define OFFSET 5
#define SIZE 12

#define PAUSE 2

int trial = 1;
int fd, pid;
char buffer[120], myname[20];
void writeAccess(), readAccess();
void fcleanup(int nsig){
    unlink(DataFile);
    printf("cleanup:%s\n", myname);
    if(nsig) exit(0);
}

int main(){
    int i;

    fd = creat(DataFile, 0644);
    write(fd, info, strlen(info));
    close(fd);

    signal(SIGINT, fcleanup);

    sprintf(myname, fork() ? "B-%06d" : "A-%06d", pid = getpid());

    srand(time(NULL)+pid);
    printf("%s:started\n", myname);

    fd = open(DataFile, O_RDWR|O_EXCL);
    printf("%s:opened %s\n", myname, DataFile);

    for(i=0; i < 30; i++){
        if(rand()%2) readAccess();
        else writeAccess();
    }

    close(fd);

    printf("%s:finished\n", myname);
}
```

```

        wait(NULL);
        fcleanup(0);
        return 0;
    }
    void writeAccess(){
        flock_t lock;

        printf("Write:%s #%d\n", myname, trial);

        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = (off_t) OFFSET;
        lock.l_len = (size_t) SIZE;

        if(fcntl(fd, F_SETLKW, &lock) <0)
            perror("F_SETLKW");
        printf("\twrite:%s locked\n", myname);

        sprintf(buffer, "%s #%02d", myname, trial);
        printf ("\twrite:%s \"%s\"\n", myname, buffer);

        lseek (fd, (off_t) OFFSET, SEEK_SET);
        write (fd, buffer, SIZE);

        sleep (PAUSE);

        lock.l_type = F_UNLCK;
        if(fcntl(fd, F_SETLKW, &lock) <0)
            perror("F_SETLKW");

        printf("\twrite:%s unlocked\n", myname);

        trial++;
    }
    void readAccess(){
        flock_t lock;

        printf("Read:%s #%d\n", myname, trial);

        lock.l_type = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = (off_t) OFFSET;
        lock.l_len = (size_t) SIZE;

        if(fcntl(fd, F_SETLKW, &lock) <0)
            perror("F_SETLKW");
        printf("\tread:%s locked\n", myname);

        lseek(fd, (off_t) OFFSET, SEEK_SET);
        read (fd, buffer, SIZE);

        printf("\tcontents:%s \"%*.s\"\n", myname, SIZE, SIZE, buffer);
        sleep (PAUSE);

        lock.l_type = F_UNLCK;
        if(fcntl(fd, F_SETLKW, &lock) <0)
            perror("F_SETLKW");

        printf("\tread:%s unlocked\n", myname);

        trial++;
    }
}

```

Исследуя выдачу этой программы, вы можете обнаружить, что READ-области могут перекрываться; но что никогда не перекрываются области READ и WRITE ни в какой комбинации. Если идет чтение процессом А – то запись процессом В дожидается разблокировки А (чтение – не будет дожидаться). Если идет запись процессом А – то и чтение процессом В и запись процессом В дождутся разблокировки А.

6.9.2. UNIX SVR4 имеет еще один интерфейс для блокировки файлов: функцию *lockf*.

```

#include <unistd.h>

int lockf(int fd, int operation, size_t size);

```

Операция **operation**:

F_ULOCK

Разблокировать указанный сегмент файла (это может снимать один или несколько замков).

F_LOCK

F_LOCK

Установить замок. При этом, если уже имеется чужой замок на запрашиваемую область, *F_LOCK* блокирует процесс, *F_LOCK* – просто выдает ошибку (функция возвращает -1, *errno* устанавливается в *EAGAIN*).

- Ожидание отпирания/запирания замка может быть прервано сигналом.
- Замок устанавливается следующим образом: от текущей позиции указателя чтения/записи в файле **fd** (что не похоже на *fcntl*, где позиция задается явно как параметр в структуре); длиной **size**. Отрицательное значение **size** означает отсчет от текущей позиции к началу файла. Нулевое значение – означает "от текущей позиции до конца файла". При этом "конец файла" понимается именно как конец, а не как текущий размер файла. Если файл изменит размер, запертая область все равно будет простирается до конца файла (уже нового).
- Замки, установленные процессом, автоматически отпираются при завершении процесса.

F_TEST

Проверить наличие замка. Функция возвращает 0, если замка нет; -1 в противном случае (заперто).

Если устанавливается замок, перекрывающийся с уже установленным, то замки объединяются.

```
было:      #####
запрошено: #####
стало:      #####
```

Если снимается замок с области, покрывающей только часть заблокированной прежде, остаток области остается как отдельный замок.

```
было:      #####
запрошено: _XXXXXXXXXX_
стало:      ###      ###
```

6.10. Файлы устройств.

Пространство дисковой памяти может состоять из нескольких **файловых систем** (в дальнейшем FS), т.е. логических и/или физических дисков. Каждая файловая система имеет древовидную логическую структуру (каталоги, подкаталоги и файлы) и имеет **свой** корневой каталог. Файлы в каждой FS имеют свои собственные I-узлы и собственную их нумерацию с 1. В начале каждой FS зарезервированы:

- блок для загрузчика – программы, вызываемой аппаратно при включении машины (загрузчик записывает с диска в память машины программу */boot*, которая в свою очередь загружает в память ядро */unix*);
- **суперблок** – блок заголовка файловой системы, хранящий размер файловой системы (в блоках), размер блока (512, 1024, ...), количество I-узлов, начало списка свободных блоков, и другие сведения об FS;
- некоторая непрерывная область диска для хранения I-узлов – "I-файл".

Файловые системы объединяются в единую древовидную иерархию операций **монтирования** подключения корня файловой системы к какому-то из каталогов-"листьев" дерева другой FS.

Файлы в объединенной иерархии адресуются при помощи двух способов:

- имен, задающих путь в дереве каталогов:

```
/usr/abs/bin/hackIt
bin/hackIt
./../bin/vi
```

(этот способ предназначен для **программ**, пользующихся файлами, а также пользователей);

- внутренних адресов, используемых программами ядра и некоторыми системными программами.

Поскольку в каждой FS имеется **собственная** нумерация I-узлов, то файл в объединенной иерархии должен адресоваться **ДВУМЯ** параметрами:

- номером (кодом) устройства, содержащего файловую систему, в которой находится искомый файл: *dev_t i_dev*;
- номером I-узла файла в этой файловой системе: *ino_t i_number*;

Преобразование имени файла в объединенной файловой иерархии в такую адресную пару выполняет в ядре уже упоминавшаяся выше функция *namei* (при помощи просмотра каталогов):

```
struct inode *ip = namei(...);
```

Создаваемая ею копия I-узла в памяти ядра содержит поля *i_dev* и *i_number* (которые на самом диске не хранятся!).

Рассмотрим некоторые алгоритмы работы ядра с файлами. Ниже они приведены чисто схематично и в сильном упрощении. Форматы вызова (и оформление) функций не соответствуют форматам, используемым на самом деле в ядре; верны лишь названия функций. Опущены проверки на корректность, подсчет ссылок на структуры *file* и *inode*, блокировка I-узлов и кэш-буферов от одновременного доступа, и многое другое.

Пусть мы хотим открыть файл для чтения и прочесть из него некоторую информацию. Вызовы открытия и закрытия файла имеют схему (часть ее будет объяснена позже):

```
#include <sys/types.h>
#include <sys/inode.h>
#include <sys/file.h>
int fd_read = open(имяФайла, O_RDONLY){

    int fd; struct inode *ip; struct file *fp; dev_t dev;

    u_error = 0; /* errno в программе */
    // Найти файл по имени. Создается копия I-узла в памяти:
    ip = namei(имяФайла, LOOKUP);
    // namei может выдать ошибку, если нет такого файла
    if(u_error) return(-1); // ошибка

    // Выделяется структура "открытый файл":
    fp = falloc(ip, FREAD);
    // fp->f_flag = FREAD; открыт на чтение
    // fp->f_offset = 0; RWptr
    // fp->f_inode = ip; ссылка на I-узел

    // Выделить новый дескриптор
    for(fd=0; fd < NOFILE; fd++)
        if(u_ofile[fd] == NULL) // свободен
            goto done;
    u_error = EMFILE; return (-1);
done:
    u_ofile[fd] = fp;

    // Если это устройство - инициализировать его.
    // Это функция openi(ip, fp->f_flag);
    dev = ip->i_rdev;
    if((ip->i_mode & IFMT) == IFCHR)
        (*cdevsw[major(dev)].d_open)(minor(dev), fp->f_flag);
    else if((ip->i_mode & IFMT) == IFBLK)
        (*bdevsw[major(dev)].d_open)(minor(dev), fp->f_flag);
    return fd; // через u_rval1
}

close(fd){
    struct file *fp = u_ofile[fd];
    struct inode *ip = fp->f_inode;
    dev_t dev = ip->i_rdev;

    if((ip->i_mode & IFMT) == IFCHR)
        (*cdevsw[major(dev)].d_close)(minor(dev), fp->f_flag);
    else if((ip->i_mode & IFMT) == IFBLK)
        (*bdevsw[major(dev)].d_close)(minor(dev), fp->f_flag);

    u_ofile[fd] = NULL;
    // и удалить ненужные структуры из ядра.
}
```

Теперь рассмотрим функцию преобразования логических блоков файла в номера физических блоков в файловой системе. Для этого преобразования в I-узле файла содержится таблица адресов блоков. Она устроена довольно сложно – ее начало находится в узле, а продолжение – в нескольких блоках в самой файловой системе (устройство это можно увидеть в примере "Фрагментированность файловой системы" в приложении). Мы для простоты будем предполагать, что это просто линейный массив *i_addr[]*, в котором *n*-ому логическому блоку файла отвечает *bno*-тый физический блок файловой системы:

```
bno = ip->i_addr[n];
```

Если файл является интерфейсом устройства, то этот файл не хранит информации в логической файловой системе. Поэтому у устройств нет таблицы адресов блоков. Вместо этого, поле *i_addr[0]* используется для

хранения **кода устройства**, к которому приводит этот специальный файл. Это поле носит название **i_rdev**, т.е. как бы сделано

```
#define i_rdev i_addr[0]
```

(на самом деле используется union). Устройства бывают **байто-ориентированные**, обмен с которыми производится по одному байту (как с терминалом или с коммуникационным портом); и **блочно-ориентированные**, обмен с которыми возможен только большими порциями блоками (пример – диск). То, что файл является устройством, помечено в поле **тип файла**

```
ip->i_mode & IFMT
```

одним из значений: *IFCHR* – байтовое; или *IFBLK* – блочное. Алгоритм вычисления номера блока:

```
ushort u_pboff; // смещение от начала блока
ushort u_pbsize; // сколько байт надо использовать
// ushort - это unsigned short, смотри <sys/types.h>
// daddr_t - это long (disk address)

daddr_t bmap(struct inode *ip,
              off_t offset, unsigned count){
    int sz, rem;

    // вычислить логический номер блока по позиции Rwptr.
    // BSIZE - это размер блока файловой системы,
    // эта константа определена в <sys/param.h>
    daddr_t bno = offset / BSIZE;
    // если BSIZE == 1 Kб, то можно offset >> 10

    u_pboff = offset % BSIZE;
    // это можно записать как offset & 01777

    sz = BSIZE - u_pboff;
    // столько байт надо взять из этого блока,
    // начиная с позиции u_pboff.

    if(count < sz) sz = count;
    u_pbsize = sz;
```

Если файл представляет собой устройство, то трансляция логических блоков в физические не производится – устройство представляет собой "сырой" диск без файлов и каталогов, т.е. обращение происходит сразу по физическому номеру блока:

```
if((ip->i_mode & IFMT) == IFBLK) // block device
    return bno; // raw disk
// иначе провести пересчет:

rem = ip->i_size /*длина файла*/ - offset;
// это остаток файла.
if( rem < 0 ) rem = 0;
// файл короче, чем заказано нами:
if( rem < sz ) sz = rem;
if((u_pbsize = sz) == 0) return (-1); // EOF

// и, собственно, замена логич. номера на физич.
return ip->i_addr[bno];
}
```

Теперь рассмотрим алгоритм *read*. Параметры, начинающиеся с *u_...*, на самом деле передаются как статические через вспомогательные переменные в *u-area* процесса.

```
read(int fd, char *u_base, unsigned u_count){
    unsigned srccount = u_count;
    struct file *fp = u_ofile[fd];
    struct inode *ip = fp->f_inode;
    struct buf *bp;
    daddr_t bno; // очередной блок файла

    // dev - устройство,
    // интерфейсом которого является файл-устройство,
    // или на котором расположен обычный файл.
    dev_t dev = (ip->i_mode & (IFCHR|IFBLK)) ?
        ip->i_rdev : ip->i_dev;

    switch( ip->i_mode & IFMT ){

    case IFCHR: // байто-ориентированное устройство
        (*cdevsw[major(dev)].d_read)(minor(dev));
        // прочие параметры передаются через u-area
        break;
```

```

case IFREG: // обычный файл
case IFDIR: // каталог
case IFBLK: // блочно-ориентированное устройство
do{
    bno = bmap(ip, fp->f_offset /*RWptr*/, u_count);
    if(u_pbsize==0 || (long)bno < 0) break; // EOF
    bp = bread(dev, bno); // block read

    iomove(bp->b_addr + u_pboff, u_pbsize, B_READ);

```

Функция *iomove* копирует данные

```
bp->b_addr[ u_pboff..u_pboff+u_pbsize-1 ]
```

из адресного пространства ядра (из буфера в ядре) в адресное пространство процесса по адресам

```
u_base[ 0..u_pbsize-1 ]
```

то есть пересылает **u_pbsize** байт между ядром и процессом (**u_base** попадает в *iomove* через статическую переменную). При записи вызовом *write()*, *iomove* с флагом *B_WRITE* производит обратное копирование – из памяти процесса в память ядра. Продолжим:

```

// продвинуть счетчики и указатели:
u_count -= u_pbsize;
u_base += u_pbsize;
fp->f_offset += u_pbsize; // RWptr
} while( u_count != 0 );
break;
...
return( srccount - u_count );
} // end read

```

Теперь обсудим некоторые места этого алгоритма. Сначала посмотрим, как происходит обращение к байтовому устройству. Вместо адресов блоков мы получаем код устройства **i_rdev**. Коды устройств в *UNIX* (тип *dev_t*) представляют собой пару двух чисел, называемых **мажор** и **минор**, хранимых в старшем и младшем байтах кода устройства:

```

#define major(dev) ((dev >> 8) & 0x7F)
#define minor(dev) ( dev      & 0xFF)

```

Мажор обозначает **тип устройства** (диск, терминал, и.т.п.) и приводит к одному из драйверов (если у нас есть 8 терминалов, то их обслуживает один и тот же драйвер); а **минор** обозначает **номер устройства** данного типа (... каждый из терминалов имеет миноры 0..7). Миноры обычно служат индексами в некоторой таблице структур внутри выбранного драйвера. Мажор же служит индексом в переключательной таблице устройств. При этом блочно-ориентированные устройства выбираются в одной таблице – *bdevsw[]*, а байтоориентированные – в другой – *cdevsw[]* (см. *<sys/conf.h>*; имена таблиц означают **block/character device switch**). Каждая строка таблицы содержит адреса функций, выполняющих некоторые predetermined операции способом, зависящим от устройства. Сами эти функции реализованы в драйверах устройств. Аргументом для этих функций обычно служит **минор** устройства, к которому производится обращение. Функция в драйвере использует этот минор как **индекс** для выбора конкретного экземпляра устройства данного типа; как индекс в массиве управляющих структур (содержащих текущее состояние, режимы работы, адреса функций прерываний, адреса очередей данных и.т.п. каждого конкретного устройства) для данного типа устройств. Эти управляющие структуры **различны** для разных типов устройств (и их драйверов).

Каждая строка переключательной таблицы содержит адреса функций, выполняющих операции *open*, *close*, *read*, *write*, *ioctl*, *select*. *open* служит для инициализации устройства при первом его открытии (**++ip->i_count==1**) – например, для включения мотора; *close* – для выключения при последнем закрытии (**--ip->i_count==0**). У блочных устройств поля для *read* и *write* объединены в функцию *strategy*, вызываемую с параметром *B_READ* или *B_WRITE*. Вызов *ioctl* предназначен для управления параметрами работы устройства. Операция *select* – для опроса: есть ли поступившие в устройство данные (например, есть ли в *clist*-е ввода с клавиатуры байты? см. главу "Экранные библиотеки"). Вызов *select* применим только к некоторым байтоориентированным устройствам и сетевым портам (*socket*-ам). Если данное устройство не умеет выполнять такую операцию, то есть запрос к этой операции должен вернуть в программу ошибку (например, операция *read* неприменима к принтеру), то в переключательной таблице содержится специальное имя функции *nodev*; если же операция допустима, но является фиктивной (как *write* для */dev/null*) – имя *nulldev*. Обе эти функции-заглушки представляют собой "пустышки": {}.

Теперь обратимся к блочно-ориентированным устройствам. *UNIX* использует внутри ядра дополнительную **буферизацию** при обменах с такими устройствами/-. Используемая нами выше функция **bp=bread(dev,bno);** производит чтение физического блока номер **bno** с устройства **dev**. Эта операция обращается к драйверу

конкретного устройства и вызывает чтение блока в некоторую область памяти в ядре ОС: в один из **кэш-буферов** (cache, "запасать"). Заголовки кэш-буферов (`struct buf`) организованы в список и имеют поля (см. файл `<sys/buf.h>`):

b_dev

код устройства, с которого прочитан блок;

b_blkno

номер физического блока, хранящегося в буфере в данный момент;

b_flags

флаги блока (см. ниже);

b_addr

адрес участка памяти (как правило в самом ядре), в котором собственно и хранится содержимое блока.

Буферизация блоков позволяет системе экономить число обращений к диску. При обращении к `bread()` сначала происходит поиск блока (`dev, bno`) в таблице кэш-буферов. Если блок уже был ранее прочитан в кэш, то обращения к диску не происходит, поскольку копия содержимого дискового блока уже есть в памяти ядра. Если же блока еще нет в кэш-буферах, то в ядре выделяется чистый буфер, в заголовке ему прописываются нужные значения полей **b_dev** и **b_blkno**, и блок считывается в буфер с диска вызовом функции

```
bp->b_flags |= B_READ; // род работы: прочитать
(*bdevsw[major(dev)].d_strategy)(bp);
// bno и минор - берутся из полей *bp
```

из драйвера конкретного устройства.

Когда мы что-то изменяем в файле вызовом `write()`, то изменения на самом деле происходят в кэш-буферах в памяти ядра, а не сразу на диске. При записи в блок буфер помечается как **измененный**:

```
b_flags* B_DELWRI; // отложенная запись
```

и на диск немедленно не записывается. Измененные буфера физически записываются на диск в таких случаях:

- Был сделан системный вызов `sync()`;
- Ядру не хватает кэш-буферов (их число ограничено). Тогда самый старый буфер (к которому дольше всего не было обращений) записывается на диск и после этого используется для другого блока.
- Файловая система была отмонтирована вызовом `umount`;

Понятно, что **не измененные** блоки обратно на диск из буферов не записываются (т.к. на диске и так содержатся те же самые данные). Даже если файл уже закрыт `close`, его блоки могут быть еще не записаны на диск – запись произойдет лишь при вызове `sync`. Это означает, что измененные блоки записываются на диск "массированно" – по многу блоков, но не очень часто, что позволяет оптимизировать и саму запись на диск: сортировкой блоков можно достичь минимизации перемещения магнитных головок над диском.

Отслеживание самых "старых" буферов происходит за счет реорганизации списка заголовков кэш-буферов. В большом упрощении это можно представить так: как только к блоку происходит обращение, соответствующий заголовок переставляется в начало списка. В итоге самый "пассивный" блок оказывается в хвосте – он то и переиспользуется при нужде.

"Подвешивание" файлов в памяти ядра значительно ускоряет работу программ, т.к. работа с памятью гораздо быстрее, чем с диском. Если блок надо считать/записать, а он уже есть в кэше, то реального обращения к диску не происходит. Зато, если случится сбой питания (или кто-то неаккуратно выключит машину), а некоторые буфера еще не были сброшены на диск – то часть изменений в файлах будет потеряна. Для принудительной записи всех измененных кэш-буферов на диск существует сисвызов "синхронизации" содержимого дисков и памяти

```
sync(); // synchronize
```

Вызов `sync` делается раз в 30 секунд специальным служебным процессом `/etc/update`, запускаемым при загрузке системы. Для работы с файлами, которые должны гарантированно быть корректными на диске, используется открытие файла

```
fd = open( имя, O_RDWR | O_SYNC);
```

которое означает, что при каждом `write` блок из кэш-буфера **немедленно** записывается на диск. Это делает работу надежнее, но существенно медленнее.

Специальные файлы устройств не могут быть созданы вызовом `creat`, создающим только обычные файлы. Файлы устройств создаются вызовом `mknod`:


```
#include <sys/sysmacros.h>
dev_t dev = makedev(major, minor);
/* (major << 8) | minor */
mknod( имяФайла, кодыДоступа|тип, dev);
```

где **dev** – пара (мажор,минор) создаваемого устройства; **кодыДоступа** – коды доступа к файлу (0777)*; **тип** – это одна из констант *S_IFIFO*, *S_IFCHR*, *S_IFBLK* из include-файла *<sys/stat.h>*.

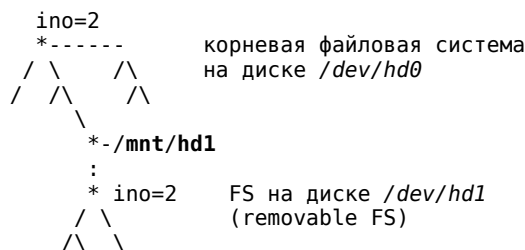
mknod доступен для выполнения только суперпользователю (за исключением случая *S_IFIFO*). Если бы это было не так, то можно было бы создать файл устройства, связанный с существующим диском, и читать информацию с него напрямую, в обход механизмов логической файловой системы и защиты файлов кодами доступа.

Можно создать файл устройства с мажором и/или минором, не отвечающим никакому реальному устройству (нет такого драйвера или минор слишком велик). Открытие таких устройств выдает код ошибки *ENODEV*.

Из нашей программы мы можем вызовом *stat()* узнать код устройства, на котором расположен файл. Он будет содержаться в поле *dev_t st_dev*; а если файл является специальным файлом (интерфейсом драйвера устройства), то код самого этого устройства можно узнать из поля *dev_t st_rdev*; Рассмотрим пример, который выясняет, относятся ли два имени к одному и тому же файлу:

```
#include <sys/types.h>
#include <sys/stat.h>
void main(ac, av) char *av[]; {
    struct stat st1, st2; int eq;
    if(ac != 3) exit(13);
    stat(av[1], &st1); stat(av[2], &st2);
    if(eq =
        (st1.st_ino == st2.st_ino && /* номера I-узлов */
         st1.st_dev == st2.st_dev)) /* коды устройств */
        printf("%s и %s - два имени одного файла\n", av[1], av[2]);
    exit( !eq );
}
```

Наконец, вернемся к склейке нескольких файловых систем в одну объединенную иерархию:



Для того, чтобы поместить корневой каталог файловой системы, находящейся на диске */dev/hd1*, вместо каталога */mnt/hd1* уже "собранный" файловой системы, мы должны издать сисвызов

```
mount("/dev/hd1", "/mnt/hd1", 0);
```

Для отключения смонтированной файловой системы мы должны вызвать

```
umount("/dev/hd1");
```

(каталог, к которому она смонтирована, уже числится в таблице ядра, поэтому его задавать не надо). При монтировании все содержимое каталога */mnt/hd1* станет недоступным, зато при обращении к имени */mnt/hd1* мы на самом деле доберемся до (безымянного) корневого каталога на диске */dev/hd1*. Такой каталог носит название **mount point** и может быть выявлен по тому признаку, что "." и ".." в нем лежат на разных устройствах:

```
struct stat st1, st2;
stat("/mnt/hd1/.", &st1); stat("/mnt/hd1/..", &st2);
if( st1.st_dev != st2.st_dev) ... ; /*mount point*/
```

Для *st1* поле *st_dev* означает код устройства */dev/hd1*, а для *st2* – устройства, содержащего корневую файловую систему. Операции монтирования и отмонтирования файловых систем доступны только суперпользователю.

И напоследок – сравнение структур I-узла.

на диске	в памяти	в вызове stat
<i><sys/ino.h></i>	<i><sys/inode.h></i>	<i><sys/stat.h></i>
struct <i>dinode</i>	struct <i>inode</i>	struct <i>stat</i>

```

// коды доступа и тип файла
ushort di_mode      i_mode      st_mode
// число имен файла
short  di_nlink     i_nlink     st_nlink
// номер I-узла
ushort ---          i_number    st_ino
// идентификатор владельца
ushort di_uid       i_uid       st_uid
// идентификатор группы владельца
ushort di_gid       i_gid       st_gid
// размер файла в байтах
off_t  di_size      i_size      st_size
// время создания
time_t di_ctime     i_ctime     st_ctime
// время последнего изменения (write)
time_t di_mtime     i_mtime     st_mtime
// время последнего доступа (read/write)
time_t di_atime     i_atime     st_atime
// устройство, на котором расположен файл
dev_t  ---          i_dev       st_dev
// устройство, к которому приводит спец.файл
dev_t  ---          i_rdev      st_rdev
// адреса блоков
char   di_addr[39]  i_addr[]
// счетчик ссылок на структуру в ядре
cnt_t  ---          i_count
//
// и кое-что еще

```

Минусы означают, что данное поле не хранится на диске, а вычисляется ядром. В современных версиях *UNIX* могут быть легкие отличия от вышенаписанной таблицы.

* - Следует отличать эту системную буферизацию от буферизации при помощи библиотеки *stdio*. Библиотека создает буфер в самом **процессе**, тогда как системные вызовы имеют буфера внутри **ядра**.

* - Обычно к блочным устройствам (дискам) доступ разрешается только суперпользователю, в противном случае можно прочесть с "сырого" диска (в обход механизмов файловой системы) физические блоки любого файла и весь механизм защиты окажется неработающим.

© Copyright A. Богатырев, 1992-95
Си в UNIX

[Назад](#) | [Содержание](#) | [Вперед](#)

[\[Главная \]](#) [\[Гостевая \]](#)

