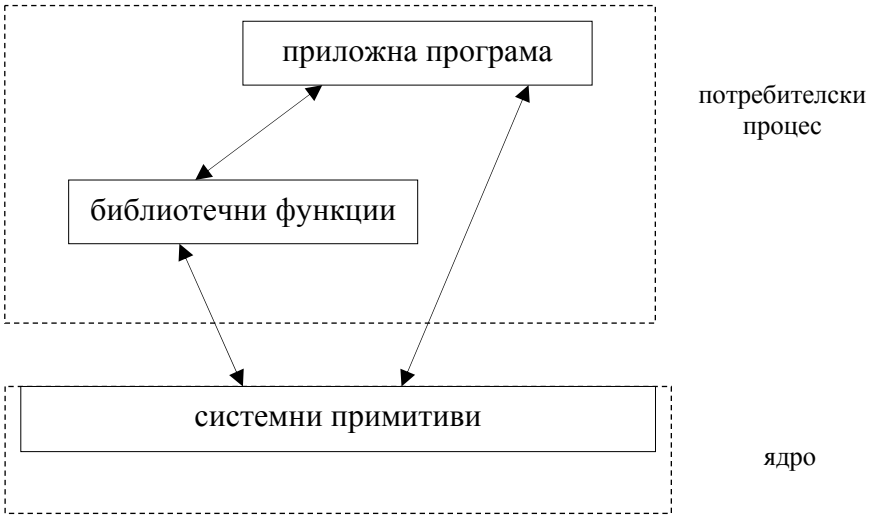
1. УВОД

Всяка операционна система (в тесен смисъл на това понятие, т.е. ядрото) осигурява определен набор от операции за изпълняваните програми. Тези операции обикновено се наричат системни извиквания, system calls, системни примитиви, системни функции. Тук ще разгледаме операциите, реализирани в различните версии на Unix и Linux системите, които са включени в стандартите POSIX.

1.1. Системни примитиви и библиотечни функции

Системните примитиви са програмният интерфейс на ядрото на ОС. В ранните версии на UNIX Version 7 те са около 50, в съвременните версии като UNIX System V, 4.4BSD или Linux те са над 100. Техниката, използвана при реализацията им, е еднаква за всички. За всеки системен примитив има поне една функция в стандартната библиотека на езика С. Потребителската програма съдържа обръщение към съответната функция. Това, което прави функцията е да предаде управлението в ядрото, като най-често използваната техника е с програмно прекъсване. Действителната работа по системните примитиви се върши от програмен код в ядрото. Системните примитиви обикновено са документирани в раздел 2 от документацията.

Библиотеките на езика С включват още много други функции, които не са входове в ядрото, но някои могат да съдържат обръщение към системни примитиви. Напр., функцията `atoi` се реализира изцяло от кода в библиотеката, а функцията `printf` извиква системния примитив `write` за писане във файла. Библиотечните функции са документирани в раздел 3 от документацията. Разликата и връзката между системните примитиви и библиотечните функции е показана на Фиг.1.1.



Фиг.1.1. Разлика между системни примитиви и библиотечни функции

1.2. Обработка на грешки

При успех функцията на системния примитив обикновено връща неотрицателно цяло. Когато се случи грешка при изпълнение на системен примитив, функцията връща отрицателно число, обикновено -1. Причините за грешка при системен примитив може да са различни и понякога в зависимост от причината е необходимо да се предприемат различни действия. Механизмът за докладване на грешки (още от най-ранните версии на Unix) е чрез глобалната променлива `errno`, чрез която ядрото връща код на грешката. Файлът `<errno.h>` съдържа дефиницията на променливата `errno` и всички значения, които тя може да приема. Кодовете на грешки са цели положителни числа, но за тях са определени символни константи. Например, ако `errno` е `ENOENT` това означава, че

файлът, чието име е зададено в примитива не съществува (обикновено се появява текста No Such File or Directory), а `EACCES` означава проблем с правата на потребителския процес (Permission denied).

При анализа на грешки от системен примитив има едно важно обстоятелство. Значението на променливата `errno` никога не се чисти от ядрото. Това означава, че трябва да я проверяваме само след грешка на системен примитив, т.е. след като функцията е върнала -1. Следният фрагмент показва как може да се прави проверка за грешка.

```
/* ------------------------------------------------------------ */
#include <errno.h>

int fd;
  . . .
  if ( (fd = open("myfile",O_RDONLY)) == -1 ) { /* Fail to open? */
     if ( errno == ENOENT ) /* myfile does not exist? */
       fd = open("Myfile",O_RDONLY); /* No, so try another file */
  }
  if ( fd == -1 ) { /* Did either open fail? */
   /* Yes, report the open failure... */
  } else {
   /* myfile or Myfile is open */
  }
  . . .
/* ------------------------------------------------------------ */
```

Когато при грешка искаме само да докладваме това на потребителя, трябва да има начин за преобразуване на кода от `errno` в подходящо съобщение, което да изведем. Съществуват няколко начина за конструиране и извеждане на подходящи съобщения.

- Чрез библиотечната функция `perror(3)`, която конструира събщение и го извежда на стандартния изход за грешки.

```
#include <stdio.h>

void perror(const char *s);
```

Извежда на стандартния изход за грешки низа `s`, последван от двоеточие и текста за текущото значение в `errno`.

- Чрез библиотечната функция `strerror(3)`, която връща текст за зададен в аргумента й код на грешка.

```
#include <string.h>

char *strerror(int errnum);
```

- Чрез библиотечната функция `fprintf(3)` за форматиран изход на стандартния изход за грешки `stderr`.

```
#include <stdio.h>

int fprintf(FILE *stream, char *frm ...);
```

Следват две програми, които представляват тест на тези функции.

Пример

Програма 1.1. извежда съобщение за грешка чрез `perror`.

```
/* ------------------------------------------------------------ */
/* Example of perror */

#include <stdio.h>
#include <errno.h>
main(void)
{
  errno = EACCES;
```

```
  perror("Test EACCES Message");
  exit(0);
}
/* ------------------------------------------------------- */
```

Ако компилираме тази програма във файл `a.out` и я изпълним, на стандартния изход за грешки ще бъде изведено следното:

```
$ a.out
Test EACCES Message: Permission denied
```

В примерите по-нататък ще използваме този начин на запис за командите, които въвеждаме и получения изход: символите, които ние въвеждаме ще са показани с **`този шрифт`**, а изхода от програмата по `този начин`. Симолът „`$`", предшестващ нашия вход е поканата (prompt), извеждан от командния интерпретатор.

Пример

Програма 1.2. извежда съобщение за грешка чрез `strerror` и `fprintf`.

```
/* ------------------------------------------------------- */
/* Example of strerror */

#include <stdio.h>
#include <errno.h>
#include <string.h>

main(void)
{
  int i;
  static int ecodes[] = { -1, EACCES, 4096 } ;

  for ( i=0; i<3; ++i ) {
    errno = ecodes[i];
    fprintf(stderr, "%4d = '%s'\n",ecodes[i], strerror(errno));
  }
  exit(0);
}
/* ------------------------------------------------------- */
```

Получихме следните резултати на стандартния изход за грешки:

```
$ a.out
  -1 = 'Unknown error -1'
  13 = 'Permission denied'
4096 = 'Unknown error 4096'
```

Тъй като значенията `-1` и `4096` са несъществуващи кодове на грешки то `strerror` връща съобщението `'Unknown error'`.

При проверка за грешки и извеждане на съобщения в примерите по-нататък ще използваме следните функции (определени са в нашия заглавен файл `ourhdr.h`).

```
void err_sys_exit(char * frm, …);
```

Извежда съобщение за грешка след грешка в системен примитив и завършва процеса.

```
void err_sys(char * frm, …);
```

Извежда съобщение за грешка след грешка в системен примитив и връща управлението.

```
void err_exit(char *frm, …);
```

Извежда съобщение за грешка, която не е свързана със системен примитив, и завършва процеса.

```
void err_ret(char *frm, …);
```

Извежда съобщение за грешка, която не е свързана със системен примитив, и връща управлението.

Функциите приемат променлив брой аргументи. Първият аргумент `frm` е форматиращ низ, съставян по същите правила както в `printf`, а останалите аргументи не са задължителни и имат същата роля както съответните аргументи в `printf`.

1.3. Стандарт POSIX и реализациите му

POSIX (от Portable Operating System Interface for Computer Environment) е фамилия от стандарти, разработени в IEEE. Някои от тях са за командния интерпретатор (shell) и обслужващите програми, за системната администрация. От интерес за нас са тези, определящи програмния интерфейс, а именно POSIX 1003.1. Тъй като, в стандарта се определя интерфейса, но не и реализацията, там не се прави разлика между системен примитив и библиотечна функция. Използва се просто термина функция.

Този стандарт има реализации в различни Unix системи. Основните клонове на Unix системи са: UNIX System V Release 4 (SVR4), 4.3BSD и Linux. Ние ще продължим да правим разлика между системен примитив и библиотечна функция, защото ще работим в средата на конкретна ОС. Примерите са тествани в различни версии на Linux, основно в Scientific Linux version 2.6.9-22.0.2.EL, а в някои случаи и други версии.

Съществуват различни вълшебни константи, които определят различни ограничения на реализацията. Ограниченията ще разделим на следните типове:

- ограничения, определени в заглавни файлове (compile-time limits)

Ако дадено ограничение е фиксирано за съответната реализация, то може да се определи във заглавен файл, това е основно `<limits.h>`.

- ограничения, определяни по време на изпълнение (run-time limits)

Този тип ограничения се определят чрез функциите `sysconf(3)` и `pathconf(3)`.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);
```

Чрез функцията `pathconf` определяме ограничения свързани с файлове и каталози, като максимална дължина на собствено име на файл, максимална дължина на пълно име на файл и др.

Чрез функцията `sysconf` определяме ограничения не свързани с файлове и каталози, като максимален брой процеси за един реален uid, максимална обща дължина на аргументите, предавани на `exec`, максимален брой отворени файлове за процес и др.

Аргументът *`name`* определя името на ограничението, и при успех функцията връща значението му или -1 при грешка.

Пример

Програма 1.3. извежда ограничения, определени в системата, като обработва и случаите, когато ограничението не е определено.

```
/* ------------------------------------------------------------ */
/* Print sysconf and pathconf values */

#include "ourhdr.h"

void pr_sysconf(char *, int);
void pr_pathconf(char *, char *, int);
```

```
main(int argc, char *argv[])
{
  if (argc != 2)
    err_exit("usage: a.out dirname");

  pr_sysconf("ARG_MAX            =", _SC_ARG_MAX);
  pr_sysconf("CHILD_MAX          =", _SC_CHILD_MAX);
  pr_sysconf("clock tics/second  =", _SC_CLK_TCK);
  pr_sysconf("OPEN_MAX           =", _SC_OPEN_MAX);
#ifdef _SC_ STREAM_MAX
  pr_sysconf("STREAM_MAX         =", _SC_STREAM_MAX);
#endif
  pr_sysconf("_POSIX_JOB_CONTROL =", _SC_JOB_CONTROL);
  pr_sysconf("_POSIX_SAVED_IDS   =", _SC_SAVED_IDS);
  pr_sysconf("_POSIX_VERSION     =", _SC_VERSION);

  pr_pathconf("LINK_MAX           =", argv[1], _PC_LINK_MAX);
  pr_pathconf("NAME_MAX           =", argv[1], _PC_NAME_MAX);
  pr_pathconf("PATH_MAX           =", argv[1], _PC_PATH_MAX);
  pr_pathconf("PIPE_BUF           =", argv[1], _PC_PIPE_BUF);
  pr_pathconf("MAX_CANON          =", "/dev/tty", _PC_MAX_CANON);
  pr_pathconf("MAX_INPUT          =", "/dev/tty", _PC_MAX_INPUT);
  pr_pathconf("_POSIX_NO_TRUNC    =", argv[1], _PC_NO_TRUNC);
  pr_pathconf("_POSIX_CHOWN_RESTRICTED =", argv[1], _PC_CHOWN_RESTRICTED);
  exit(0);
}

void pr_sysconf(char * msg, int name)
{
  long val;
  fputs(msg, stdout);
  errno = 0;
  if ( (val=sysconf(name)) < 0) {
     if (errno !=0 )
       err_sys_exit("sysconf error");
     fputs (" (not defined)\n", stdout); }
  else
     printf (" %ld\n", val);
}

void pr_pathconf(char * msg, char * path, int name)
{
  long val;
  fputs(msg, stdout);
  errno = 0;
  if ( (val=pathconf(path, name)) < 0) {
     if (errno !=0 )
       err_sys_exit("pathconf error, path=%s", path);
     fputs (" (no limit)\n", stdout); }
  else
     printf (" %ld\n", val);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата получихме следните резултати в две различни версии на Linux: в Red Hat Linux 4.1 и Scientific Linux:

/*------ в Red Hat Linux 4.1 ------*/

```
ARG_MAX            = 131072
CHILD_MAX          = 999
clock tics/second  = 100
OPEN_MAX           = 256
```

```
STREAM_MAX         = 256
_POSIX_JOB_CONTROL = 1
_POSIX_SAVED_IDS   = 1
_POSIX_VERSION     = 199309
LINK_MAX           = 127
NAME_MAX           = 255
PATH_MAX           = 1024
PIPE_BUF           = 4096
MAX_CANON          = 255
MAX_INPUT          = 255
_POSIX_NO_TRUNC    = 1
_POSIX_CHOWN_RESTRICTED = 1
```

/*------ в Scientific Linux ------*/

```
ARG_MAX            = 131072
CHILD_MAX          = 999
clock tics/second  = 100
OPEN_MAX           = 1024
STREAM_MAX         = 16
_POSIX_JOB_CONTROL = 1
_POSIX_SAVED_IDS   = 1
_POSIX_VERSION     = 200112
LINK_MAX           = 32000
NAME_MAX           = 255
PATH_MAX           = 4096
PIPE_BUF           = 4096
MAX_CANON          = 255
MAX_INPUT          = 255
_POSIX_NO_TRUNC    = 1
_POSIX_CHOWN_RESTRICTED = 1
```

1.4. Примитивни системни типове данни

Стандартът POSIX въвежда така наречените примитивни системни типове данни. Те са производни типове данни, чиито имена завършват на _t и се използват от системните примитиви. Обикновено те са дефинирани в заглавния файл `<sys/types.h>` чрез `typedef` и реализацията им е системно зависима. Като използваме тези типове, нашите програми няма да зависят от детайли на реализацията, които могат да се променят в различни POSIX операционни системи. Следва списък на основните типове, които ще използваме.

Тип	**Описание**
`dev_t`	номер на устройство (major и minor)
`gid_t`	идентификатор на група
`ino_t`	номер на i-node
`mode_t`	права на достъп до файл
`nlink_t`	брой твърди връзки към файл
`off_t`	размер на файл и текуща позиция във файл
`pid_t`	идентификатор на процес
`size_t`	размер на обект без знак
`ssize_t`	размер на обект със знак
`time_t`	календарна дата и време в брой секунди от начална точка (1.1.1970)
`uid_t`	потребителски идентификатор

Фиг.1.2. Примитивни системни типове данни

2. ВХОД И ИЗХОД ЧРЕЗ СИСТЕМНИ ПРИМИТИВИ

Ще започнем с основните системни примитиви за вход и изход, които работят с обикновен файл, като отваряне, четене, писане и др. Основното проектно решение, което определя набора от операции над файл, е структурата на файла. За потребителя файлът представлява последователност от 0 или повече байта. Това е структурата на файла, реализирана от файловата система и съобразно тази структура са проектирани системните примитиви.

2.1. Файлов дескриптор и системни таблици

Идентификатор на отворен файл се нарича **файлов дескриптор** (file descriptor). Файловият дескриптор е цяло неотрицателно число. Когато се изпълни системен примитив `open` за отваряне на файл, ядрото връща в процеса файлов дескриптор. При всички следващи операции с файла - четене, писане и др. процесът идентифицира файла чрез този файлов дескриптор. Файловият дескриптор се освобождава при изпълнение на системен примитив `close`.

Файловите дескриптори имат локално значение за процеса, т.е. за всеки процес те приемат значения от 0 до `OPEN_MAX-1` (19 в ранните версии на Unix, 63, 255, 1023 в по-новите). По традиция командният интерпретатор свързва файлов дескриптор 0 със стандартния вход, 1 със стандартния изход и 2 със стандартния изход за грешки. В заглавния файл `<unistd.h>` са определени символните константи `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`, които съответстват на трите вълшебни числа.

С всяко отваряне на файл се свързва и указател на **текуща позиция** във файла (file offset, file pointer), който определя позицията във файла, от която ще бъде четено или записвано и на практика стойността му е отместването от началото на файла, измерено в байтове. Този указател се зарежда, изменя или използва от повечето примитиви за работа с файлове.

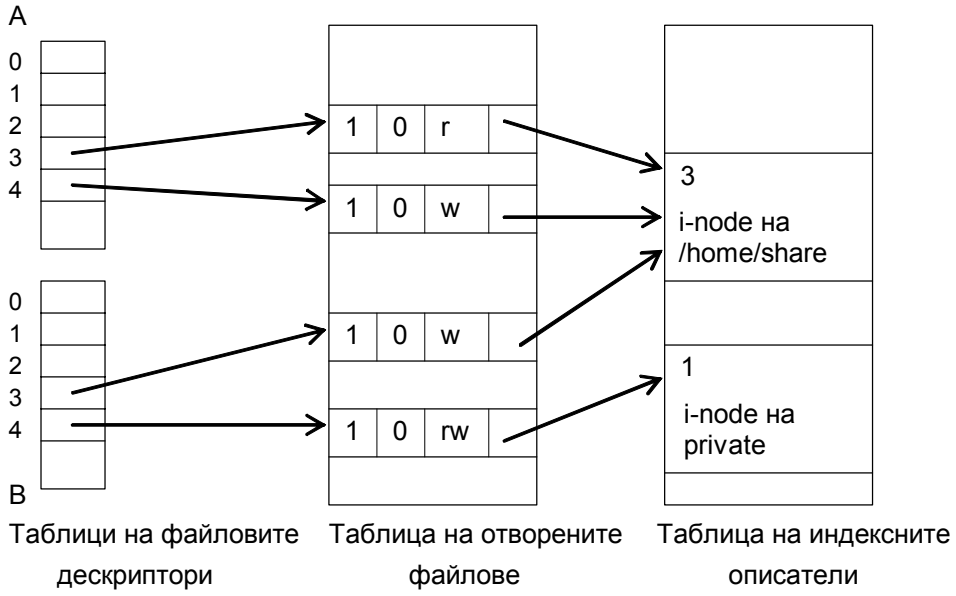
С всяко отваряне на файл (т.е. с всеки файлов дескриптор) се свързва и **режим на отваряне** на файла, който определя начина на достъп до файла докато е отворен от съответния процес, напр. само четене, само писане, четене и писане. При всеки следващ опит за достъп до файла се проверява дали той не противоречи на режима на отваряне, и ако е така, достъпът се отказва независимо от правата на процеса.

Три основни структури данни в ядрото се използват при реализацията на системните примитиви на файловата система. Наричат ги **системни таблици** и са разположени в пространството на ядрото. Това са:

1. Таблица на индексните описатели - За всеки отворен файл в таблицата на индексните описатели се пази точно един запис, съдържащ копие на индексния описател (i-node) от диска и някои други полета, като номер на устройството и номер на i-node, състояние на i-node, брояч (брой указатели, насочени към записа).
2. Таблица на отворените файлове - При всяко отваряне на файл се отделя един запис в тази таблица, който съдържа:
   - текущата позиция във файла
   - режима на отваряне на файла
   - указател към съответния запис от таблицата на индексните описатели
   - брояч (брой указатели, насочени към записа, 1 след отваряне).
3. Таблици на файловите дескриптори - Всеки процес има собствена таблица на файловите дескриптори. Броят на елементите в нея определя максималния брой едновременно отворени файлове за процес (`OPEN_MAX`). Записите в тези таблици съдържат само указател към запис от таблицата на отворените файлове и флагове

на файловия дескриптор. Файловият дескриптор всъщност представлява индекс на използвания при отварянето запис от таблицата на файловите дескриптори.

Фигура 2.1 представя връзката между записите в тези три системни таблици, ако два процеса, означени като A и B са отворили един и същи файл. Процес A е отворил файл `/home/share` веднаж за четене и веднаж за писане. Процес B е отворил файл `/home/share` за четене и файл `private` за четене и писане.



Фиг. 2.1. Системни таблици на файловата система

При всяко отваряне на файл се създава верига в системните таблици от файловия дескриптор до индексния описател, с което се осигурява достъп на процеса до данните на файла независимо от останалите процеси.

2.2. Системни примитиви `open` и `creat`

Файл се отваря или създава с примитива `open`.

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

int open(const char *filename, int oflag [, mode_t mode]);
```

Връща файлов дескриптор при успех, -1 при грешка.

```
int creat(const char * filename, mode_t mode);
```

Връща файлов дескриптор само за писане при успех, -1 при грешка.

Аргументът *`filename`* задава името на файла. В по-ранните версии на Unix примитивът `open` е предназначен само за отваряне на съществуващ файл, т.е. създава връзка между процеса и указания файл, която се идентифицира чрез файлов дескриптор върнат от `open` и включва режима на отваряне на файла и текуща позиция във файла. След `open` текущата позиция сочи началото на файла. В по-новите версии на Unix и в Linux функциите на `open` са разширени. Чрез `open` може да се създаде файл, ако не съществува и след това да се отвори. Аргументът *`mode`* определя кода на защита на новосъздадения файл. Функциите на системния примитив се конкретизират чрез

параметъра *oflag*. Значенията на *oflag* се конструират чрез побитово ИЛИ (|) от следните символни константи (определени в <fcntl.h>):

- Определящи начина на достъп (трябва да се зададе само един от тези три флага):

O_RDONLY - (0) четене
O_WRONLY - (1) писане
O_RDWR - (2) четене и писане

- Определящи действия, извършвани при изпълнение на open:

O_CREAT - създаване на нов файл, ако не съществува
O_EXCL - (заедно с O_CREAT) връща грешка, ако файлът съществува
O_TRUNC - изменяне дължината на файла на 0, ако съществува

- Определящи действия, извършвани при писане във файла:

O_APPEND - добавяне в края на файла при всяка операция писане
O_SYNC - синхронно обновяване на данните на диска при всяко извикване на системния примитив write.

Някои от флаговете в режима на отваряне за определен файлов дескриптор могат да бъдат изменени впоследствие чрез системния примитив fcntl.

Нов файл може да бъде създаден и чрез примитива creat. Примитивът creat създава нов файл с указаното име *filename* и код на защита *mode*. Ако такъв файл вече съществува, то старото му съдържание се унищожава при условие, че процесът има право за писане във файла (ще го наричаме право w). Процесът трябва да има право за търсене (ще го наричаме право x) за всички каталози на пътя в пълното име на файла и право w за родителския каталог. Същите права се изискват и при създаване на файл чрез open. Създаденият файл се отваря само за писане и creat връща файлов дескриптор свързан с файла.

Следователно, системният примитив creat е излишен (но е запазен заради съвместимост с по-ранните версии и може би за удобство), тъй като е еквивалентен на

```
open(filename, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Един недостатък на creat е, че новосъзданеният файл се отваря само за писане. Ако искаме да създадем временен файл, в който първо ще пишем, а след това ще четем, то трябва да изпълним creat, close и след това open. В този случай по-добре е да използваме:

```
open(filename, O_RDWR|O_CREAT|O_TRUNC, mode)
```

Един файл може да бъде отворен едновременно от няколко процеса и дори няколко пъти от един процес. При всяко отваряне процесът, изпълняващ open, получава нов файлов дескриптор, с който са свързани независими текуща позиция и режим на отваряне. Всеки open или creat разпределя нов запис в съответната таблица на файловите дескриптори и нов запис в таблицата на отворените файлове.

Фиг.2.1 изобразява вида на таблиците след като процес А изпълни:

```
fd1 = open("/home/share", O_RDONLY);
fd2 = open("/home/share", O_WRONLY);
```

а процес В изпълни системните примитиви:

```
fd1 = open("/home/share", O_WRONLY);
fd2 = open("private", O_RDWR);
```

Защо тогава има отделна структура - Таблица на отворените файлове? От всичко казано до сега се вижда, че съответствието между записите в Таблицата на файловите

дескриптори и Таблицата на отворените файлове е едно към едно. Но това е за сега. Системните примитиви `dup` и `fork`, които ще разгледаме по-нататък, позволяват няколко записа от таблиците на файловите дескриптори да сочат към един запис от таблицата на отворените файлове.

2.3. Системен примитив `close`

Примитивът `close` затваря отворен файл, т.е. прекратява връзката между процеса и файла, като освобождава файловия дескриптор, който след това може да бъде използван при последващи изпълнения на `creat` или `open`.

```
#include <unistd.h>
int close(int fd);
```

Връща 0 при успех, -1 при грешка.

Когато процес завършва (изпълнява `exit`), ядрото автоматично затваря всички отворени файлове. Затова понякога в програмите не се извиква явно `close` за затваряне на файловете.

2.4. Системни примитиви `read` и `write`

Примитивът `read` чете *`nbytes`* последователни байта от файла, идентифициран чрез файлов дескриптор *`fd`*, като започва четенето от текущата позиция. Прочетените байтове се записват в област на процеса, чийто адрес е зададен в *`buffer`*. Указателят на текуща позиция се увеличава с действителния брой прочетени байта, т.е. сочи байта след последния прочетен байт. Връща действителния брой прочетени байта, който може да е по-малък от *`nbytes`*, 0 ако текущата позиция е след края на файла (EOF) или -1 при грешка.

```
#include <unistd.h>
ssize_t read(int fd, void *buffer, size_t nbytes);
```

Връща брой прочетени байта при успех, 0 при EOF, -1 при грешка.

```
ssize_t write(int fd, void *buffer, size_t  nbytes);
```

Връща брой записани байта при успех, -1 при грешка.

Има няколко случая когато действителният брой прочетени байта може да е по-малък от *`nbytes`*:

- При четене от обикновен файл, ако до края на файла има по-малко байта. Ще прочете толкова байта, колкото има до края на файла. Следващия `read` ще върне 0.
- При четене от специален файл за терминал, обикновено се чете един ред.
- При четене от програмен канал, чете докато прочете искания брой байта или колкото има в момента в канала, което от двете се случи първо.

Примитивът `write` има същите аргументи като `read`. *`Nbytes`* байта се предават от областта на процеса с адрес *`buffer`* към файла, идентифициран с файлов дескриптор *`fd`*. Аналогично на `read`, мястото на началото на писане във файла се определя от текущата позиция, като след завършване на обмена текущата позиция се увеличава с броя записани байта, т.е. се премества след последния записан байт. За разлика от `read`, `write` може да пише и в позиция след края на файла, при което се извършва увеличаване размера на файла. Ако е вдигнат флаг `O_APPEND` при `open`, то преди всяко изпълнение на `write`

текущата позиция се установява в края на файла. Ако е вдигнат флаг `O_SYNC` при `open`, то `write` връща управлението след физическото записване на данните и управляващата информация на диска.

Изпълнявайки `read` или `write` в цикъл след `open` може последователно да се прочете или запише файл, т.е. системните примитиви, разгледани до тук осигуряват последователен достъп до файл.

Пример

Програма 2.1 копира стандартния вход на стандартния изход.

```
/* ------------------------------------------------------------ */
/* Copy standart input to standart output */

#include "ourhdr.h"
#define BUFFSIZE 4096

main(void)
{
  int n;
  char buf[BUFFSIZE];

  while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
       err_sys_exit("write error");

  if (n < 0)
     err_sys_exit("read error");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако искаме да копираме файл `xx` във файл `yy`, то ще извикаме програмата:

```
$ a.out < xx > yy
```

Един въпрос, свързан с този пример, е как сме избрали значението на `BUFFSIZE`. Значението на `BUFFSIZE` би могло да е всяко едно цяло положително число, включително и 1. Ние сме избрали това значение, защото размерът на блок във файловата система на Linux, в която работим, е 4096. Тогава постигаме най-добро системно време за изпълнение. По-нататъшното увеличаване на размера на програмния буфер няма да има съществен ефект.

2.5. Системен примитив `lseek`

След `open` текущата позиция във файла сочи началото му, те е 0, освен ако не е зададен флаг `O_APPEND`. Всеки `read` или `write` премества автоматично текущата позиция с действителния брой прочетени или записани байта. Примитивът `lseek` позволява да се премести указателя на текуща позиция на произволна позиция във файла или след края му без входно/изходна операция.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int flag);
                          Връща новата текуща позиция при успех, -1 при грешка.
```

Аргументът *`offset`* задава отместването, с което текущата позиция ще се промени, а *`flag`* определя началото от което се отчита отместването: 0 - от началото на файла, 1 - от текущото значение на указателя, 2 - от края на файла. В заглавния файл `<unistd.h>` са

дефинирани три символни константи за значението на аргумента *flag*. Следователно новото значение на текущата позиция във файла се изчислява в зависимост от значението на *flag* по следния начин:

```
0 (SEEK_SET)     fp = offset
1 (SEEK_CUR)     fp = fp + offset
2 (SEEK_END)     fp = file_size + offset
```

При изпълнението на примитива `lseek` не се изисква достъп до диска. Изменя се единствено полето за текуща позиция в съответния запис от таблицата на отворените файлове. Това значение ще бъде използвано при последващото четене или писане. Значението в *offset* може да е положително или отрицателно число. Ако новото значение е след EOF, то размерът на файла не се увеличава. Това ще стане по-късно, при изпълнение на последващия `write`. Ако за ново значение се получи отрицателно число, то това се счита за грешка и текущата позиция не се изменя. (За някои типове специални файлове текущата позиция може да е отрицателно цяло. Затова при проверка за грешка при `lseek`, трябва да сравняваме с -1.)

Тъй като при успех `lseek` връща новото значение на текущата позиция, то можем да използваме `lseek` за да определим текущата позиция без да я изменяме:

```
off_t curpos;
curpos = lseek(fd, 0, SEEK_CUR);
```

Този начин на извикване можем да използваме и за проверка дали файлът позволява позициониране. Някои типове файлове, като програмните канали, специалния файл за терминал, не позволяват позициониране чрез `lseek` и тогава примитивът връща -1.

Пример

Програма 2.2 проверява дали стандартния вход може да бъде позициониран.

```
/* ------------------------------------------------------------ */
/* Test if standart input is capable of seeking */

#include "ourhdr.h"

main(void)
{
  if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
    printf("cannot seek\n");
  else
    printf("seek OK\n");

  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълним програмата ще получим следните резултати:

```
$ a.out < /etc/passwd
seek OK
$ cat /etc/passwd | a.out
cannot seek
$ a.out < /dev/tty
cannot seek
```

Ако новото значение на текущата позиция след `lseek` е на разстояние след края на файла, то последващия `write` ще започне писането от текущата позиция и ще увеличи размера на файла. Така може да се създаде „файл с дупка”. При четене всички байтове от дупката са нулеви (\0).

Пример

Програма 2.3 създава „файл с дупка”.

```
/* ------------------------------------------------------------ */
/* Create a file with a hole */

#include "ourhdr.h"

char buf1[] = "ABCDEF";
char buf2[] = "abcdef";

main(void)
{
  int fd;
  if ((fd = creat("file.hole", 0640)) < 0)
    err_sys_exit("create error");

  if (write(fd, buf1, 6) != 6)
    err_sys_exit("write buf1 error");

  if (lseek(fd, 10, SEEK_CUR) == -1)
    err_sys_exit("lseek error");

  if (write(fd, buf2, 6) != 6)
    err_sys_exit("write buf2 error");

  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълним програмата ще получим следните резултати:

```
$ a.out
$ ls -l file.hole
-rw-r-----  1 moni staff 22 Apr 12 05:58 file.hole
$ od -c file.hole
0000000   A   B   C   D   E   F  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020   a   b   c   d   e   f
0000026
```

Използваме командата `od` с аргумент `-c`, за да видим съдържанието на файла по символи. От изхода се вижда, че десетте незаписани байта в средата се четат като `\0`.

Използването на `read` или `write` съвместно с `lseek` осигурява произволен достъп до файл.

2.6. Неделимост на операциите

Под неделима или атомарна операция се разбира операция, която се състои от няколко стъпки, но се осигурява изпълнението им по принципа „всичко или нищо”, т.е или всички стъпки се изпълняват или нито една от тях. Ядрото гарантира неделимост за всеки отделен системен примитив. Ако няколко процеса са отворили един и същи файл и четат и пишат в него, могат ли да възникнат проблеми. Да разгледаме случая, при който един процес ще добавя в края на файл. Един вариант за това е:

```
if (lseek(fd, 0L, SEEK_END) == -1) /* position to EOF */
    err_sys_exit("lseek error");
if (write(fd, buf, 10) != 10)      /* and write */
    err_sys_exit("write error");
```

Това ще работи без проблем, ако само един процес е отворил файла. Но ще възникне проблем ако няколко процеса използват тази техника за да пишат в края на един и същи файл. Нека процесите A и B от Фиг.2.1 са отворили файла `/home/share` за писане и

изпълняват тази последователност от `lseek` и `write`. Да предположим, че двата процеса изпълнят примитивите в следната последователност.

Процес А	Време	Процес В
`lseek`	t1	
	t2	`lseek`
	t3	`write`
`write`	t4	

При такава последователност процесът А пише върху данните записани от процеса В.

Проблемът тук се състои в това, че логическата операция „позициониране в края на файла и писане" се реализира чрез два системни примитива. Да си спомним, че ядрото гарантира неделимост за всеки отделен системен примитив. Докато не завърши изпълнението на примитива `write`, `lseek` или друг, съответният запис в таблицата на индексните описатели е заключен и друг процес не може да започне изпълнение на примитив над този файл. Но всяка операция, която изисква повече от един примитив не е атомарна. Решението тук е да осигурим неделимост на двете операции – позициониране и писане. В по-новите версии на Unix и в Linux това е възможно, като зададем флаг `O_APPEND` в `open`. Тогава ядрото позиционира в края на файла при всеки `write` и не е нужно да изпълняваме `lseek`.

Друг пример за проблем при взаимодействието на няколко процеса, осъществяващи достъп до един и същи файл, е показан в следния фрагмент:

```
if ((fd = open("file", O_WRONLY)) < 0)
   if (errno == ENOENT) {
       if ((fd = creat("file", 0640)) < 0)
           err_sys_exit("create error");
   } else
       err_sys_exit("open error");
```

В този фрагмент проверката за съществуването на файла и създаването му, ако не съществува, се извършват с два системни примитива. Това означава, че между `open` и `creat` е възможно друг процес да създаде файла и ако този процес запише нещо във файла, тези данни ще бъдат изтрити при `creat`. В по-новите версии на `open` можем да използваме флагове `O_CREAT` и `O_EXCL`, за да осигурим неделимост на операцията „проверка дали файл съществува и създаването му ако не съществува".

2.7. Системни примитиви `dup` и `dup2`

Съществуващ файлов дескриптор може да бъде копиран (дублиран) чрез един от примитивите `dup` и `dup2`.

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int newfd);
```

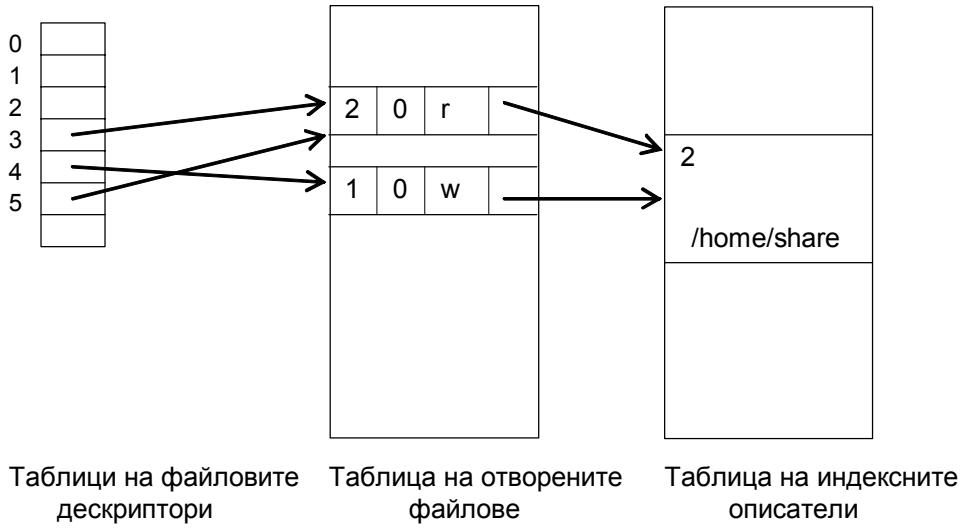
Връща нов файлов дескриптор при успех, -1 при грешка.

При `dup` файловият дескриптор в аргумента *`fd`* се копира на първото свободно място в Таблицата на файловите дескриптори. При `dup2` чрез аргумента *`newfd`* се определя мястото в Таблицата на файловите дескриптори, където да се копира *`fd`*. Ако *`newfd`* е отворен файлов дескриптор, то първо се затваря, след което там се копира *`fd`*. Ако *`fd`* и *`newfd`* са еднакви, то `dup2` връща *`newfd`* без да го затваря и да копира.

Какво е общото между двата файлови дескриптори - *fd* и копието, върнато от функцията? Да предположим, че процес е изпълил системните примитиви:

```
fd1 = open("/home/share", O_RDONLY);
fd2 = open("/home/share", O_WRONLY);
fd3 = dup(fd1);
read(fd1, buf1, sizeof(buf1));
read(fd3, buf2, sizeof(buf2));
```

На Фиг.2.2 е показано съдържанието на съответните записи в системните таблици. От там се вижда, че полето брояч в съответния запис от таблицата на отворените файлове е увеличено с 1. Също така става ясно, че двата буфера `buf1` и `buf2` няма да съдържат еднакви, а последователни данни на файла. Ако след това се изпълни `close(fd1)`, четенето от файла може да продължи нормално чрез `fd3`.



Фиг 2.2. Системните таблици след `dup`

Следователно двата файлови дескриптора имат следното общо:
- един и същи отворен файл;
- общ указател на текуща позиция;
- еднакъв режим на отваряне на файла.

Примитивът `dup` се използва при пренасочване на стандартен вход, изход или изход за грешки и при организиране на конвейер от програми. Следва програмен фрагмент за пренасочване на стандартен вход от файл:

```
if ( (fd = open(infile, O_RDONLY)) == -1 )
     err_sys_exit("can't open file %s\n", infile);
close(0);
dup(fd);
close(fd);
```

Друг начин да се копира файлов дескриптор е чрез системния примитив `fcntl`. Всъщност действието на:

```
dup(int fd);
```

е еквивалентно на

```
fcntl(fd, F_DUPFD, 0);
```

както и действието на:

```
dup2(int fd, newfd);
```

е еквивалентно на

```
close(newfd);
fcntl(fd, F_DUPFD, newfd);
```

Във втория случай има разлика, която е свързана с атомарността на операциите. При варианта с `dup2` операцията е атомарна, докато алтернативният начин е с два системни примитива. Между `close` и `fcntl` процесът може да получи сигнал, за който е предвидил обработка и обработчика да промени файловите дескриптори.

2.8. Системен примитив `fcntl`

С този примитив могат да се изменят различни свойства на отворен файл.

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, long arg);
```

Връща в зависимост от *cmd* при успех, -1 при грешка.

Ще разгледаме следните основни приложения на `fcntl`:

- копиране на файлов дескриптор
- четене/изменение на флагове на файлов дескриптор
- четене/изменение на режим на отваряне.

При тези операции третият аргумент е от тип `long`. Операцията се определя чрез аргумента *cmd*, който има следните значения:

`F_DUPFD` Копира файловия дескриптор *fd* на първото свободно място в Таблицата на файловите дескриптори, което е равно или по-голямо от *arg*. Връща новия файлов дескриптор. Следователно *fd* и новият файлов дескриптор разделят общ запис в Таблицата на отворените файлове (както при `dup`), но за новия файлов дескриптор флагът `FD_CLOEXEC` е свален. Това означава, че при следващ `exec` този файлов дескриптор ще остане отворен.

`F_GETFD` Връща флаговете на файловия дескриптор *fd* като значение на функцията. За сега се реализира само флаг `FD_CLOEXEC`. Ако е вдигнат този флаг при следващ `exec` файловият дескриптор ще бъде затворен, а ако е свален - файловият дескриптор ще стане отворен.

`F_SETFD` Изменя флаговете на файловия дескриптор *fd*. Новото значение се взема от аргумента *arg*.

`F_GETFL` Връща флаговете от режима на отваряне за файлов дескриптор *fd* като значение на функцията. Тези флагове разглеждахме при `open`.

`F_SETFL` Изменя флаговете от режима на отваряне за файлов дескриптор *fd*. Новото значение се взема от аргумента *arg*. Могат да се изменят само флагове: `O_APPEND`, `O_NONBLOCK`, `O_SYNC`.

Пример

Програма 2.4 приема един аргумент, който счита за файлов дескриптор и извежда описание на флаговете в режима на отваряне.

```
/* ------------------------------------------------------- */
/* Print (file) access flags for file descriptor */

#include <fcntl.h>
#include "ourhdr.h"
```

```
main(int argc, char *argv[])
{
  int val, accmode;
  if (argc !=2)
     err_exit("usage: a.out file_descriptor");

  if ((val=fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
     err_sys_exit("fcntl error for fd %d", atoi(argv[1]));

  accmode = val & O_ACCMODE;
  if (accmode == O_RDONLY)      printf("read only");
  else if (accmode == O_WRONLY) printf("write only");
  else if (accmode == O_RDWR)   printf("read write");
  else err_exit("unknown access mode");
  if (val & O_APPEND)     printf(", append");
  if (val & O_NONBLOCK)   printf(", nonblocking");
  if (val & O_SYNC)       printf(", synchronous write");

  putchar('\n');
  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата в средата на bash на Linux получихме следните резултати:

```
$ a.out 0 < /dev/tty
read only
$ a.out 1 > temp
$ cat temp
write only
$ a.out 2 2>>temp
write only, append
$ a.out 4 4<>temp
read write
```

В този команден интерпретатор конструкцията `4<>temp` се интерпретира като отваряне на файл `temp` за четене и писане във файлов дескриптор 4.

Примитивът `fcntl` работи с файлов дескриптор, без да му е нужно да знае името на файла. Това е полезно, например когато искаме на изменим флаг в режима на отваряне на стандартния вход или изход. Тези файлове процесът получава отворени и не знае името на файла, което се крие зад съответния файлов дескриптор. Например, нека в Пример 2.1 искаме да използваме синхронно писане при копирането на файла, т.е. всеки `write` да чака данните да бъдат записани на диска преди да върне управлението. В началото преди цикъла трябва да вдигнем флаг `O_SYNC`, като добавим следния код:

```
int val;
if (( val=fcntl(STDOUT_FILENO, F_GETFL, 0)) < 0
   err_sys_exit("fcntl  F_GETFL error");
val != O_SYNC;
if ( fcntl(STDOUT_FILENO, F_SETFL, val) < 0 )
   err_sys_exit("fcntl  F_SETFL error");
```

3. АТРИБУТИ НА ФАЙЛ

Ще продължим със системни примитиви на файловата система, които работят с атрибутите на файл. Ще започнем с примитива `stat`, и разглеждане на различните атрибути на файл, а след това и другите системни примитиви, чрез които се изменят значенията на някои от атрибутите на файл.

3.1. Системни примитиви `stat`, `fstat` и `lstat`

Част от съхраняваната за файл информация в индексния му описател може да се получи чрез примитивите `stat`, `fstat` и `lstat`.

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

int stat(const char *filename, struct stat *sbuf);

int fstat(int fd, struct stat *sbuf);

int lstat(const char *filename, struct stat *sbuf);
```

Връщат 0 при успех, -1 при грешка.

Първият аргумент идентифицира файла чрез име *`filename`* при `stat` и `lstat`, или чрез файлов дескриптор *`fd`* при `fstat`. Действието на примитива `lstat` се различава от това на `stat` и `fstat`, когато файлът е от тип символна връзка. Тогава при `lstat` се връща информация за файла-символна връзка, а не се следва символната връзка. Примитивът `fstat` е полезен при работа с наследени отворени файлове, чиито имена може да не са известни на процеса. Процесът трябва да има право `x` за всички каталози на пътя в пълното име на файла, но не се изискват никакви права за самия файл.

Вторият аргумент *`sbuf`* е указател на структура `stat`, в която примитивът записва информацията за файла. Дефиницията на структурата може леко да се различава в различните реализации, но често изглежда така:

```
struct stat {
   dev_t st_dev;           /* device where inode belongs */
   ino_t st_ino;           /* inode number */
   mode_t st_mode;         /* mode word */
   nlink_t st_nlink;       /* number of hard links */
   uid_t st_uid;           /* user ID of owner*/
   gid_t st_gid;           /* group ID of owner */
   dev_t st_rdev;          /* device type for special files */
   off_t st_size;          /* file size */
   blksize_t st_blksize;   /* blocksize for filesystem I/O */
   blkcnt_t st_blocks;     /* number of blocks allocated */
   time_t st_atime;        /* time of last access */
   time_t st_mtime;        /* time of last modification */
   time_t st_ctime;        /* time of last change */
};
```

Повечето елементи в тази структура съответстват на атрибутите на файл, които се съхраняват в индексния му описател. Изключение правят `st_dev`, съдържащ номер на устройството и `st_ino`, съдържащ номер на i-node в това устройство, т.е. двата елемента заедно представляват адрес на i-node във файловата система.

Ще разгледаме атрибутите на файл и свързаните с тях системни примитиви като преминем през елементите на структурата `stat`.

3.2. Типове файлове

В елемента `st_mode` са кодирани два атрибута на файла – типа и кода на защита на файла. Типът на файла се съхранява в старшите 4 бита на `st_mode`. Да си припомним основните типове файлове, реализирани в съвременните Unix и Linux системи.

1. Обикновен файл (regular file). Файл, съдържащ данни в някакъв формат.
2. Каталог (directory). Файл, съдържащ записи за други файлове, които съдържат собственото име на файл и номера на i-node му. Чрез този тип се релизира йерархичната организация на файловата система.
3. Символен специален файл (character special device file). Всеки файл от този тип съответства на символно устройство, например терминал, принтер и др.
4. Блоков специален файл (block special device file). Всеки файл от този тип съответства на блоково устройство, например диск.
5. Символна връзка (symbolic link или soft link). Символната връзка е файл, който сочи към друг файл.
6. Програмен канал (pipe, FIFO file). Тип файл, реализиращ механизъм за междупроцесни комуникации.
7. Сокет (socket). Тип файл, реализиращ механизъм за междупроцесни комуникации в мрежова среда.

Типа на файл можем да определим по-удобно чрез макроси, определени във файла `<sys/stat.h>` и показани на Фиг.3.1. Като аргумент в макроса се задава елемента `st_mode` на структура `stat`.

Макрос	**Тип на файл**
`S_ISREG()`	обикновен
`S_ISDIR()`	каталог
`S_ISCHR()`	символен специален
`S_ISBLK()`	блоков специален
`S_ISFIFO()`	програмен канал
`S_ISLNK()`	символна връзка
`S_ISSOCK()`	сокет

Фиг. 3.1. Макроси за определяне на тип на файл в `<sys/stat.h>`

Пример

Програма 3.1 приема произволен брой аргументи, които са имена на файлове, и за всеки проверява типа на файла и извежда подходящ текст. Вместо примитива `stat` използваме `lstat` за да можем да определим кой файл е символна връзка, а не да я следваме.

```
/* ------------------------------------------------------------ */
/* Print the type of file for each of the command line arguments */

#include <sys/stat.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  int i;
  extern int errno;
  struct stat statbuf;
  char *ptr;

  for (i=1; i<argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &statbuf) < 0) {
```

```
        err_sys("lstat error", argv[i]);
        continue; }

    if      (S_ISREG(statbuf.st_mode))    ptr = "regular";
    else if (S_ISDIR(statbuf.st_mode))    ptr = "directory";
    else if (S_ISCHR(statbuf.st_mode))    ptr = "character special";
    else if (S_ISBLK(statbuf.st_mode))    ptr = "block special";
#ifdef S_ISLNK
    else if (S_ISLNK(statbuf.st_mode))    ptr = "symbolic link";
#endif
#ifdef S_ISFIFO
    else if (S_ISFIFO(statbuf.st_mode))   ptr = "fifo";
#endif
#ifdef S_ISSOCK
    else if (S_ISSOCK(statbuf.st_mode))   ptr = "socket";
#endif
    else      ptr = "** unknown file type **";
    printf("%s\n", ptr);
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от програмата:

```
$ a.out a.out /etc /dev/tty /dev/hdb1 /dev/lp0 /dev/null
a.out: regular
/etc: directory
/dev/tty: character special
/dev/hdb1: block special
/dev/lp0: character special
/dev/null: character special
```

Друг начин за определяне на типа на файл е да отделим първите 4 бита от елемента `st_mode` чрез маска и след това да сравняваме с кодовете за различните типове. Във файла `<sys/stat.h>` са дефинирани символни константи: `S_IFMT` за маската и `S_IFxxx` за типовете. Следователно в оператора `if` условията ще са от вида:

```
(statbuf.st_mode & S_IFMT ) == S_IFREG
```

3.3. Код на защита на файл и права на достъп

С всеки файл е свързан потребител – собственик на файла и потребителска група, към която принадлежи собственика. Тези два атрибута са съответно в елементите `st_uid` и `st_gid`. (Там са числовите идентификатори на потребител и потребителска група.) Младшите 12 бита от `st_mode` представляват кода на защита на файла и определят правата на потребителите за достъп до този файл. Различават се три типа достъп до файл – четене, писане и изпълнение (означавани със символите `r, w, x` в изхода на командата `ls -l`), които се прилагат спрямо всеки един файл независимо от типа му. Значението на битовете в кода на защита и маските, определени в `<sys/stat.h>`, са показани на Фиг.3.2.

Осмичен код	Маска	Значение
04000	S_ISUID	Изменение на UID при изпълнение (set UID)
02000	S_ISGID	Изменение на GID при изпълнение (set GID)
01000	S_ISVTX	Sticky bit
00400 00200 00100 00700	S_IRUSR S_IWUSR S_IXUSR S_IRWXU	четене за собственика писане за собственика изпълнение за собственика четене, писане и изпълнение за собственика
00040 00020 00010 00070	S_IRGRP S_IWGRP S_IXGRP S_IRWXG	четене за групата писане за групата изпълнение за групата четене, писане и изпълнение за групата
00004 00002 00001 00007	S_IROTH S_IWOTH S_IXOTH S_IRWXO	четене за другите писане за другите изпълнение за другите четене, писане и изпълнение за другите

Фиг. 3.2. Значение на битовете в кода на защита и маски в <sys/stat.h>

Какво означават трите типа достъп за различните типове файлове.

За обикновен файл:

- Право r означава правото да отворим файла за четене, т.е. с флаг O_RDONLY или O_RDWR в open.
- За да отворим файл в режим O_WRONLY или O_RDWR трябва да имаме право w. (за режима O_RDWR е необходимо да имаме права r и w)
- Право x е необходимо за да извикаме файл за изпълнение с примитив exec.

За каталог:

- Право r означава правото да четем съдържанието на каталога. Например, за изпълнение на командата ls -l dir се изисква право r за каталога dir.
- Право w за каталог означава правото да създаваме или унищожаваме файлове от произволен тип в каталога. Например, за изпълнение на командата rm dir/text се изисква право w за dir.
- Право x означава търсене на файлове в каталога и позициониране в каталог. Например, за командите cat dir/text и cd dir се изисква право x за dir.

Правата r и x при каталози действат независимо, x не изисква r и обратно, следователно при комбинирането им могат да се получат интересни резултати. Например, каталог с право x и без r за дадена категория потребители е така наречения "тъмен каталог". Потребителите имат право да четат файловете в каталога, само ако им знаят имената. Този метод се използва в FTP сървери, когато някои раздели от архива трябва да са достъпни само за “посветени” потребители.

Следователно, най-общите правила за използване на правата на достъп при различните операции над файлове са:

- За да отворим файл с open трябва да имаме право x за всички каталози по пътя към файла, а за самия файл право, отговарящо на режима на отваряне.
- За да създадем файл в определен каталог е необходимо да имаме право x за всички каталози по пътя към файла и право w за родителския каталог.
- За да изтрием файл е необходимо да имаме право x за всички каталози по пътя към файла и право w за родителския каталог. Не са ни нужни никакви права за самия файл.

- За да се позиционираме в определен каталог трябва да имаме права `x` за всички каталози по пътя към новия текущ каталог, включително и за него.

Например, за да изпълним примитива:

```
open("/usr/prog/file1", O_RDWR);
```

трябва да имаме права: `x` за `/`, `x` за `/usr`, `x` за `/usr/prog`, `r` и `w` за `file1`.

Но какво точно означава израза „трябва да имаме права„? Когато процес извика системен примитив за отваряне, създаване, изтриване на файл и т.н. ядрото проверява правата му относно файла. При тази проверка се използват атрибутите на файла – `st_uid`, `st_gid`, `st_mode` и атрибутите на процеса. С всеки процес са свързани четири потребителски идентификатори:

- реален потребителски идентификатор (`ruid`)
- ефективен потребителски идентификатор (`euid`)
- реален идентификатор на потребителска група (`rgid`)
- ефективен идентификатор на потребителска група (`egid`)

Реалните идентификатори определят потребителя, който е стартирал процеса (в чиято сесия се изпълнява процеса). По ефективните идентификатори се определят правата на процеса. И така проверката, която ядрото изпълнява следва следните четири стъпки.

1. Ако `euid` на процеса е 0 (root), то достъпът се разрешава.
2. Ако `euid` на процеса е еднакъв с този на собственика на файла (`st_uid`), то:
   - ако в кода на защита на файла битовете за съответния тип достъп за собственика са вдигнати, достъпът се разрешава
   - иначе не се разрешава.
3. Ако `egid` на процеса е еднакъв с този на групата на файла (`st_gid`), то:
   - ако в кода на защита на файла битовете за съответния тип достъп за групата са вдигнати, достъпът се разрешава
   - иначе не се разрешава.
4. Ако в кода на защита на файла битовете за съответния тип достъп за другите са вдигнати, достъпът се разрешава, иначе не се разрешава.

Битът `S_ISVTX` в кода на защита, наречен Sticky bit (показва се като `t` при `ls -l`), има интересна история. В ранните версии на Unix този бит е бил използван при файлове, съдържащи изпълним код на често използвани програми. Когато такава програма се извика за изпълнение за първи път, копие от образа на процеса остава в свопинг областта. Това позволява по-бързо зареждане в паметта при следващи извиквания на програмата. В съвременните Unix системи, реализиращи виртуална памет, тази техника вече не е нужна. Сега този бит се използва при каталози. Ако този бит не е вдигнат за каталог, е достатъчно процесът да има право `w` за каталога, за да може да унищожи всеки файл в него. Но ако битът е вдигнат за каталог, то процес може да унищожи файл от каталога ако има право `w` за каталога и е едно от трите:

- собственик на файла
- собственик на каталога
- принадлежи на администратора (root)

Така чрез Sticky bit може да се осигури допълнителна защита на файловете в каталога. Пример за каталог с вдигнат Sticky bit е `/tmp`, в който всички могат да създават файлове, но всеки може да изтрива само собствените си файлове.

3.4. Системни примитиви `umask`, `chmod` и `fchmod`

След като разгледахме значението на битовете в кода на защита на файл и приложението им при основните операции над файл, да видим кои са примитивите свързани с кода на защита.

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

mode_t umask(mode_t cmask);
```

Връща старото значение на маската при успех.

Чрез прмитива `umask` се зарежда маската за създаване на файл за процеса. Аргументът *`cmask`* се конструира като побитово или на деветте константи от Фиг.3.2 и новата маска е *`cmask`*`&0777`. Това е един от малкото примитиви, който не връща грешка.

Тази маска влияе на примитивите, изпълнявани от процеса за създаване на файлове - `creat`, `open`, `mkdir` и `mknod` като модифицира кода на защита на създаваните файлове. Действителният код на защита при създаване е: *`mode&~cmask`*, т.е. от значение са младшите 9 бита на *`cmask`* и битовете, които са 1 в *`cmask`* стават 0 в действителния код на защита, независимо от значението им в аргумента *`mode`* на системния примитив `creat`, `open`, `mkdir` или `mknod`. Маската се наследява при пораждане на процеси. Системният примитив връща старото значение на маската, което може да бъде съхранено и по-късно възстановено. Така този примитив дава възможност на потребителите да ограничават по премълчаване достъпа до своите файлове, но само по време на създаването им.

Пример

Програма 3.2 създава два файла, един с маска 0 и един с маска, отнемаща правата `r` и `w` от групата и другите. Имената на създаваните файлове се задават като аргументи при извикване на програмата.

```
/* ------------------------------------------------------------ */
/* Example of umask */

#include <sys/stat.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  umask(0);
  if (argc != 3)
     err_exit("usage: a.out file1 file2");

  if (creat(argv[1], S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH) < 0)
     err_sys_exit("creat error for %s", argv[1]);

  umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

  if (creat(argv[2], S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH) < 0)
     err_sys_exit("creat error for %s", argv[2]);

  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от програмата.

```
$ umask
0022
```

```
$ a.out f1 f2
$ ls -l f1 f2
-rw-rw-r--  1 moni staff 0 Apr 12 05:23 f1
-rw-------  1 moni staff 0 Apr 12 05:23 f2
$ umask
0022
```

Повечето командни интерпретатори в Unix и Linux имат вътрешна команда `umask`, с която се изменя маската или се извежда нейното значение. Ние използвахме тази команда за да изведем значението на маската преди и след изпълнението на програмата. Изходът показва, че въпреки изменението на маската в програмата (на 066), след изпълнението на програмата тя има същото значение както преди това. Това се обяснява с факта, че маската е атрибут на процеса, който се наследява при пораждане на процеси (от баща към син), но не се връща обратно (от син към баща). Когато процесът завърши в родителския shell процес маската е непроменена.

Пример

Програма 3.3 извежда на стандартния изход текущата маска на процеса (аналог на командата `umask`, но без аргумент, т.е. само извежда маската без да я изменя).

```
/* ------------------------------------------------------------ */
/* Example of print umask */

#include <sys/types.h>
#include "ourhdr.h"

main(void)
{
  mode_t old_umask;

  old_umask = umask(0);
  printf("umask = %04o\n",old_umask);
  umask(old_umask);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата получаваме изход, който е същия като от командата `umask`.

```
$ a.out
umask = 0022
$ umask
0022
```

Системните примитиви, чрез които се изменя кода на защита на файл след създаването му са `chmod` и `fchmod`.

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

int chmod(const char *filename, mode_t mode);

int fchmod(int fd, mode_t mode);
```

Връщат 0 при успех, -1 при грешка.

Примитивът `chmod` изменя кода на защита на файла *`filename`* според указаното в аргумента *`mode`*. Аргументът *`mode`* се конструира като побитово ИЛИ на константи от Фиг.3.2. Примитивът `fchmod` има същото действие, но файлът се идентифицира чрез файлов дескриптор *`fd`*.

Процесът, който изпълнява `chmod` или `fchmod` трябва да принадлежи на собственика на файла или на привилегирования потребител (`euid` на процеса да е еднакъв със значението в `st_uid` или да е 0) и трябва да има права `x` за всички каталози в пълното име *`filename`*. При успех връща 0, а при грешка -1 и кода на защита не се променя.

Пример

Програма 3.4 приема един аргумент, който е име на файл и променя кода му на защита: отнема право `w` от групата и дава право `r` за другите.

```
/* ------------------------------------------------------------ */
/* Example of chmod */

#include <sys/stat.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  struct stat statbuf;

  if (argc != 2)
     err_exit("usage: a.out file");

  if (stat(argv[1], &statbuf) < 0)
    err_sys_exit("stat error for %s", argv[1]);

/* turn off group-write and turn on other-read */
  if (chmod(argv[1], (statbuf.st_mode & ~ S_IWGRP) | S_IROTH) < 0)
    err_sys_exit("chmod error for %s", argv[1]);

  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата с аргумент името на файла `f1`, който създадохме с Програма 3.2 получаваме следното:

```
$ ls -l f1
-rw-rw-r--  1 moni staff 0 Apr 12 05:23 f1
$ a.out f1
$ ls -l f1
-rw-r--r--  1 moni staff 0 Apr 12 05:23 f1
```

3.5. Системни примитиви `chown`, `fchown` и `lchown`

Всеки файл има атрибути собственик на файла и потребителска група, към която принадлежи собственика (елементите `st_uid` и `st_gid` в структурата `stat`). Когато се създава файл с `creat`, `open`, `mkdir` или `mknod`, той трябва да получи значения за тези два атрибута (за тях няма аргументи в примитивите). Собственик на файла става ефективния потребителски идентификатор на процеса (`euid`). По отношение на групата на файла има различни реализации:

- Групов идентификатор на файла става ефективния групов идентификатор на процеса (`egid`).
- Групов идентификатор на файла става груповия идентификатор на родителския му каталог. Новосъздаденият файл наследява групата от родителския си каталог. Това е подхода в 4.3 BSD.

С примитивите `chown` и `fchown` се изменя собственика и/или групата на файл след неговото създаване.

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *filename, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

int lchown(const char *filename, uid_t owner, gid_t group);
```

Връщат 0 при успех, -1 при грешка.

Трите функции имат подобно действие - изменят собственика и групата на файла според указаното в аргументите *owner* и *group*. Ако значението на аргумента *owner* или *group* е -1, то не се изменя съответно собственика или групата. Разликата между трите функции е първо в начина по който се идентифицира файла, при chown и lchown чрез име *filename*, а при fchown чрез файлов дескриптор *fd*. Освен това lchown изменя собственика и групата на самата символна връзка, а не на файла към който тя сочи.

За да се измени собственика на файл в 4.3BSD, процесът трябва да принадлежи на привилегирования потребител. В SVR4 е разрешено и всеки потребител да изменя собственика на файловете си. Според POSIX са разрешени и двата варианта в зависимост от значението на символната константа _POSIX_CHOWN_RESTRICTED, която обикновено е определена в заглавния файл <unistd.h> и значението й може да се провери с функцията pathconf (виж Програма 1.3).

Промяна на групата на файл е успешна, ако е изпълнено едно от двете:

- Процесът принадлежи на привилегирования потребител.
- Процесът принадлежи собственика на файла и *group* е равно на egid на процеса или собственикът на файла е член на групата *group*.

Освен това процесът трябва да има права x за всички каталози в пълното име *filename*. При успех връща 0, а при грешка -1 и собствеността не се променя.

3.6. Размер на файл

Елементът st_size на структурата stat съответства на поле от индексния описател и съдържа размера на файла в брой байтове, когато типът на файла е обикновен, каталог, символна връзка и програмен канал. При обикновен файл е възможно елементът да е 0, което означава, че още първия read ще върне 0 (EOF). При каталози значението никога не е 0 и обикновено е кратно на числа, като 512, 1024, и т.н. При програмни канали съдържанието е брой байтове, които се съдържат в канала. При символни връзки значението е брой байтове в името на файла, към който сочи връзката, напр.:

```
lrwxrwxrwx  1 moni staff 6 Apr 12 05:23 f1link -> dir/f1
```

В съвременните версии на Unix и Linux структурата stat съдържа още два елемента st_blksize и st_blocks. Полето st_blksize съдържа препоръчвания размер на блок при четене и писане във файла. Полето st_blocks е размер на файла в брой блокове по 512 байта, т.е. по него може да се определи действителния размер на дисковата памет, разпределена за файла.

В раздел 2.5 разглеждахме така наречения „файл с дупка”. Програма 2.3 илюстрира създаването на такъв файл. Нека сме изпълнили тази програма и сме създали файл file.hole, но дупката е с размер 8192 байта.

```
$ ls -l file.hole
-rw-r-----  1 moni staff 8204 Oct 11 14:01 file.hole
$ du file.hole
2     file.hole
```

Размерът на файла, показан с командата `ls`, е `8204`, а командата `du` показва `2` блока по 1024 байта. Командата `ls` използва полето `st_size`, а `du` използва полето `st_blocks`. Очевидно този файл съдържа голяма дупка. Когато `read` чете от дупката, се връщат нулеви байтове. Следователно, ако изпълним:

```
$ wc -c file.hole
   8204 file.hole
```

ще видим, че командата `wc` чете и брои нормално байтовете от дупката. Ако обаче направим копие на файла с командата `cat`, всички тези дупки ще бъдат реално записани като нулеви байтове.

```
$ cat file.hole > file.hole.copy
$ ls -l  file.hole*
-rw-r-----  1 moni staff 8204 Oct 11 14:01 file.hole
-rw-r-----  1 moni staff 8204 Oct 11 14:08 file.hole.copy
$ du file.hole*
2     file.hole
9     file.hole.copy
```

3.7. Времена на файл - системен примитив `utime`

Всеки файл има три атрибута за време, които се съхраняват в i-node и се намират в елементите `st_atime`, `st_mtime` и `st_ctime` на структурата `stat`. Всяко от тези полета съдържа дата и време на определено събитие от живота на файла:

- `st_atime` - на последния достъп до файла
- `st_mtime` - на последното изменение на данните на файла
- `st_ctime` - на последното изменение на i-node на файла

Забележете разликата между `st_mtime` и `st_ctime`. В тази глава разгледахме редица примитиви, които изменят атрибути на файла без да изменят данните му, напр., `chmod` и `chown`. Тези атрибути се съхраняват в индексния описател. Времето на последното подобно изменение се съхранява в `st_ctime`. Не се съхранява време на последен достъп до i-node, затова примитивът `stat` не изменя никое от трите времена.

Времето `st_atime` често се използва от администратора, за да изтрие файлове, до които не е осъществяван достъп дълго време. Типичен пример е да се изтрият файловете с име `a.out` или `core`, до които не е осъществяван достъп от месец.

Времето `st_mtime` или `st_ctime` може да се използва за да се архивират само файловете, които са били изменяни.

На Фиг.3.3 е обобщен ефекта на различните примитиви върху трите времена. Някои от функциите ще бъдат разгледани в следващите раздели.

Командата `ls` извежда само едно от тези времена. По премълчаване при опция `-l` извежда времето на последно изменение на данните. При опция `-u` извежда времето на последен достъп, а при опция `-c` времето на последното изменение на i-node на файла.

Системният примитив `utime` позволява да се изменя значението на две от времената: на последен достъп и на последно изменение на данните на файла (`st_atime` и `st_mtime`).

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *filename, const struct utimbuf *times);
```

Връща 0 при успех, -1 при грешка.

Структурата, използвана в прототипа на функцията, е определена в `<utime.h>`:

```
struct utimbuf {
   time_t actime;    /* access time */
   time_t modtime;   /* modification time */
};
```

Значението на двете времена в структурата е цяло число – брой секунди от началото на Unix епохата, която започва в 00:00:00 на 01.01.1970. Първият аргумент *filename* определя името на файла, за който ще се променят времената. Действието на примитива зависи от значението на втория аргумент.

1. Ако аргументът *times* е указател NULL, то и двете времена се изменят с времето от системния часовник. За успешното изпълнение е необходимо euid на процеса да е равен на собственика на файла или процесът да има право за писане във файла.
2. Ако аргументът *times* е различен от указател NULL, то времената се изменят със значенията на двата елемента от структурата, сочена от него. В този случай euid на процеса трябва да е равен на собственика на файла или да е 0.

Във функцията не се задава значение за времето на последно изменение на i-node на файла, защото то автоматично се изменя когато се извиква този примитив.

Пример

Програма 3.5 приема аргументи, който са имена на файлове. Съдържанието на всеки файл се освобождава (open с флаг O_TRUNC), но без да се променят времената на последен достъп и последно изменение на данните.

```
/* ------------------------------------------------------------ */
/* Example of utime */

#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  int i;
  struct stat statbuf;
  struct utimbuf timebuf;

  for (i = 1; i < argc; i++) {
    if (stat(argv[i], &statbuf) < 0) {
       err_ret("%s: stat error", argv[i]);
       continue;
    }
    if (open(argv[i], O_RDWR|O_TRUNC) < 0) {
       err_ret("%s: open error", argv[i]);
       continue;
    }
    timebuf.actime = statbuf.st_atime;
    timebuf.modtime = statbuf.st_mtime;
    if (utime(argv[i], &timebuf) < 0) {
       err_ret("%s: utime error", argv[i]);
       continue;
    }
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ще демонстрираме работата на програмата със следната последователност:

```
$ ls -l etalon xx
-rw-rw-r--  1 moni staff  293 Apr 12 01:03 etalon
```

```
-rw-rw-r--  1 moni staff 1351 Apr 12 01:47 xx
 $ ls -lu etalon xx
-rw-rw-r--  1 moni staff  293 Apr 12 07:15 etalon
-rw-rw-r--  1 moni staff 1351 Apr 18 23:17 xx
$ date
Wed Apr 20 13:21:44 EET DST 2005
$ a.out etalon xx
$ ls -l etalon xx
-rw-rw-r--  1 moni staff 0 Apr 12 01:03 etalon
-rw-rw-r--  1 moni staff 0 Apr 12 01:47 xx
$ ls -lu etalon xx
-rw-rw-r--  1 moni staff 0 Apr 12 07:15 etalon
-rw-rw-r--  1 moni staff 0 Apr 18 23:17 xx
$ ls -lc etalon xx
-rw-rw-r--  1 moni staff 0 Apr 20 13:22 etalon
-rw-rw-r--  1 moni staff 0 Apr 20 13:22 xx
```

Както очаквахме, времената на последен достъп и последно изменение на данните на двата файла не са променени, но времето на последно изменение на i-node е променено с времето на изпълнение на програмата.

Функция	Файл или каталог аргумент на функцията			Родителски каталог на файла - аргумент			Забележки
	a	m	c	a	m	c	
chmod, fchmod			•				
chown, fchown			•				
creat	•	•	•		•	•	O_CREAT на нов файл
creat		•	•				O_TRUNC на съществуващ
exec	•						
lchown			•				
link			•		•	•	
mkdir	•	•	•		•	•	
open	•	•	•		•	•	O_CREAT на нов файл
open		•	•				O_TRUNC на съществуващ
pipe	•	•	•				
read	•						
rmdir					•	•	
unlink			•		•	•	
utime	•	•	•				
write		•	•				

Фиг.3.3. Ефект на системните примитиви върху времената на файл

4. ИЗГРАЖДАНЕ НА СТРУКТУРАТА НА ФАЙЛОВАТА СИСТЕМА

В тази глава се разглеждат системни примитиви, чрез които се изгражда и манипулира системата от каталози, създават се и се унищожават връзки към файл и се изгражда единната йерархична структура на файловата система. За тази цел нека първо си припомним физическата структура на файлова система в Unix и Linux. В същност има различни реализации на файлова система в тези операционни системи, най-популярните са традиционната UNIX System V (s5fs), UFS в BSD и ext2/ext3 в Linux. Въпреки различията в детайлите, има доста общо между тези физически структури.

- Всеки файл има индексен описател или i-node (точно един), който е с фиксирана дължина (64 или 128 байта). Индексните описатели на всички файлове са разположени в специална индексната област на диска и се адресират чрез номер на i-node (пореден номер на i-node в индексната област, започвайки от 1).
- В индексния описател се съхраняват всички атрибути на файла: тип, код на защита, собственик, група, размер на файл в брой байтове и в брой блокове, дати на последен достъп до файла, последно изменение на файла и последно изменение на атрибутите на файла и др. По-голямата част от информацията за файла в структурата `stat` се получава от i-node.
- В i-node се съхраняват и определен брой адреси на дискови блокове, разпределени за файла. Тези блокове съдържат данни или адресна информация за файла (косвени блокове).
- Каталогът е тип файл, който съдържа записи за файлове. Всеки запис описва един файл и съдържа само собственото име на файла и номера на неговия i-node.
- Всеки каталог съдържа два стандартни записа, с име ".", съответстващо на самия каталог и с име ".." , съответстващо на родителския му каталог. Чрез тях се реализира йерархичната организация на файловата система.

4.1. Създаване и унищожаване на връзки към файл - `link`, `unlink` и `symlink`

Предназначението на връзките е да се позволи достъп към един файл чрез различни имена, евентуално разположени в различни каталози на файловата система. Реализират се два вида връзки към файл – твърда връзка (hard link) и символна връзка (symbolic link или soft link).

```
#include <unistd.h>
int link(const char *oldname, const char *newname);
                                        Връща 0 при успех, -1 при грешка.
```

Примитивът `link` създава нова твърда връзка за съществуващ файл, като добавя нов запис за файла в каталог. Аргументът *`oldname`* задава име на съществуващ файл, а *`newname`* - новото име на файла. Ако вече съществува файл с име *`newname`*, това е грешка. Това, което всъщност извършва `link` е да включи нов запис в родителския каталог на *`newname`* с новото име и номер на i-node от записа за *`oldname`*. Освен това в индексния описател на файла се увеличава с 1 броя на твърдите връзки за файла. Двете стъпки – добавянето на записа и изменението на i-node, се изпълняват като неделима операция. При успешно завършване двете имена на файла са напълно равноправни и на практика не може да се отличи старото от новото име.

В по-ранните версии на Unix примитивът `link` е бил разрешен и за каталози, което е било необходимо при създаване на каталози с примитива `mknod`. В по-новите версии тази необходимост не съществува, тъй като има примитив `mkdir` и затова `link` не е

разрешен за каталози. Процесът трябва да има права `x` за всички каталози в пълното име *`newname`* и *`oldname`*, и право `w` за родителския каталог на *`newname`*.

```
#include <unistd.h>
int unlink(const char *filename);
```

Връща 0 при успех, -1 при грешка.

Указаното име на файл се изключва от файловата система, т.е. изключва се записа за *`filename`* от родителския му каталога. Освен това в индексния описател на файла се намалява с 1 броя на твърдите връзки за файла. Ако броят стане 0, т.е. това е било последното име за файла, то той се унищожава (освобождават се блоковете с данни, индексния описател и косвените блокове). Ако по време на изпълнението на `unlink` файлът е отворен, то действителното му унищожаване се извършва когато последният процес, в който файлът е бил отворен, го затвори (при изпълнение на `close`).

Тази особеност в действието на `unlink` може да се използва при временни файлове. Програмният фрагмент показва как може да се организира работата с временен файл, който трябва винаги да се унищожава при завършване на процеса, дори и при аварийното му завършване.

```
main()
{
     . . .
     fd = open(temporary, O_RDWR | O_CREAT | O_TRUNC, mode);
     unlink(temporary);
     . . .
      /* четене-запис във временния файл чрез файловия дескриптор fd */
     close(fd);  /* унищожаване на временния файл */
}
```

В по-ранните версии на Unix `unlink` е бил разрешен и за каталог, в по-новите не е, тъй като там има `rmdir`, но е разрешен за символна връзка, специален файл и FIFO файл. Ако файлът е символна връзка, то `unlink` изключва символната връзка, а не файла към който тя сочи. Процесът трябва да има права `x` за всички каталози в пълното име *`filename`* и право `w` за родителския му каталог. Ако за родителския каталог е вдигнат Sticky бита, то освен това процесът трябва да е с `euid` равен на 0, или на собственика на *`filename`*, или на собственика на родителския каталог.

Пример

Програма 4.1 отваря файл след което веднага изпълнява `unlink` за него. Изчаква 20 секунди и завършва.

```
/* ------------------------------------------------------------ */
/* Example of unlink */

#include <fcntl.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  if (argc != 2)
     err_exit("usage: a.out file");

  if (open(argv[1], O_RDWR) < 0 )
     err_sys_exit("open error for %s", argv[1]);

  if (unlink(argv[1]) < 0)
```

```
     err_sys_exit("unlink error for %s", argv[1]);
  printf("file %s unlinked\n", argv[1]);
  sleep(20);
  printf("done\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ ls -l temp
-rwxr-xr-x   1 moni     staff        5864 Oct 11 13:08 temp
$ df /home
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hdb1            1946253 1432617   413040     78%   /sec
$ a.out temp &
[1] 2356
file temp unlinked
$ ls -l temp
ls: temp: No such file or directory
$ df /home
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hdb1            1946253 1432617   413040     78%   /sec
$ done
$ df /home
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hdb1            1946253 1432611   413046     78%   /sec
```

Програмата изпълняваме във фонов режим, за да можем да проверим съществуването на файла преди завършване на процеса. Според командата `ls` файлът не съществува, но според командата `df` дисковото пространство, заемано от файла не е освободено. При второто изпълнение на `df` се вижда промяна в информацията за дисковото простанство.

Пример

Програма 4.2 приема произволен брой аргументи, които са имена на файлове и се опитва да ги изтрие. Тази програма е примитивен вариант на командата `rm`.

```
/* ------------------------------------------------------------ */
/* Example of rm command */

#include <sys/stat.h>
#include "ourhdr.h"
main(int argc, char *argv[])
{
  struct stat sbuf;
  int i;

  if (argc < 2)
     err_exit("usage: a.out file...");

  for (i=1; i<argc; i++) {
    if (lstat(argv[i], &sbuf) == -1) {
       err_sys("stat error for %s", argv[i]);
       continue; }
    if ( S_ISDIR(sbuf.st_mode) ) {
       err_ret("%s is directory", argv[i]);
       continue; }

    if ( unlink(argv[i]) == -1)
       err_sys("unlink error for %s", argv[i]);
   }
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ ls -ld xx dir1 file1
ls: xx: No such file or directory
drwxr-xr-x  2 moni staff 4096 Jun  7 12:47 dir1
-rw-r--r--  1 moni staff    0 Jun  7 12:49 file1
$ a.out xx dir1 file1
stat error for xx: No such file or directory
dir1 is directory
```

Символната връзка е тип файл, който представлява „символен указател" към друг файл. Този вид връзки са въведени по-късно в 4.2BSD, заради ограниченията при твърдите връзки:

- При твърдите връзки двете имена трябва да са в една файлова система.
- Не могат да се създават твърди връзки към каталог или специален файл.

Тези ограничения не съществуват при символните връзки. Символна връзка може да се създава през границите на файловата система към обикновен файл, специален файл и към каталог. Системният примитив за създаване на символна връзка е `symlink`.

```
#include <unistd.h>

int symlink(const char *toname, const char *fromname);
```

Връща 0 при успех, -1 при грешка.

Примитивът `symlink` създава нов файл от тип символна връзка с име *fromname*, който сочи към файл *toname*, т.е. съдържанието на новия файл е низа *toname*. При твърдата връзка се гарантира съществуването на файла и след като оригиналното име е унищожено, докато при символната връзка това не е така. В същност дори не се проверява съществуването на файл *toname* при създаване на символната връзка. Символната връзка се интерпретира при опит за достъп до файла чрез нея. Процесът трябва да има права `x` за всички каталози в пълното име *fromname*, и право `w` за родителския каталог на *fromname*. Грешка е и ако вече съществува файл с име *newname*.

Когато отваряме файл с `open`, ако аргументът е име на файл от тип символна връзка, то се следва символната връзка и се отваря файлът, към който тя сочи. Ако соченият файл не съществува, то `open` връща грешка. Това може да изглежда объркващо, ако не се разбират добре символните връзки. Например:

```
$ ln -s /no/such/file temp
$ ls temp
temp
$ cat temp
cat: temp: No such file or directory
$ ls -l temp
lrwxrwxrwx  1 moni staff 13 Apr 12 07:08 temp -> /no/such/file
```

Първо създаваме файл `temp`, който е символна връзка към `/no/such/file`, чрез командата `ln -s`. Командата `ls` казва, че файл `temp` съществува. Командата `cat` казва, че файл `temp` не съществува, защото той е символна връзка, която сочи към несъществуващ файл. Командата `ls -l` ни показва две неща, че: `temp` е символна връзка (символът `l` в началото) и сочи към файл `/no/such/file` (името след ->).

Тъй като примитивът `open` следва символната връзка, то ако искаме да отворим самия файл символна връзка и да прочетем съдържанието му, трябва да използваме `readlink`. Той комбинира действието на `open`, `read` и `close`.

```
#include <unistd.h>
int readlink(const char *filename, char *buf, size_t bufsize);
```
Връща брой прочетени байта при успех, -1 при грешка.

Аргументът *filename* трябва да е име на символна връзка. Аргументът *bufsize* има същото предназначение, както третия аргумент на read. Чете най-много *bufsize* байта от съдържанието на файла в *buf* и функцията връща действителния брой прочетени байта. Ако дължината на името, към което сочи символната връзка, е по-голяма от *bufsize*, то чете само искания брой байта и това не е грешка. Процесът трябва да има права x за всички каталози в пълното име *filename*.

За примитивите, които приемат аргумент – име на файл, е важно да знаем дали следват символната връзка или не (в случай, че името е на символна връзка). Фиг. 4.1 обобщава тази информация за основните функции, които разглеждаме в раздели 2, 3 и 4.

Функция	**Не следва** символна връзка	**Следва** символна връзка
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
mkdir		•
mknod		•
open		•
opendir		•
pathconf		•
readlink	•	
rename	•	
stat		•
unlink	•	

Фиг. 4.1. Символни връзки и системни примитиви

4.2. Създаване и унищожаване на каталог - mkdir и rmdir

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *dirname, mode_t mode);
```
Връща 0 при успех, -1 при грешка.

Примитивът mkdir създава нов празен каталог с име *dirname* и код на защита *mode*. Празен каталог означава, че той съдържа само двата стандартни записа с име ".", съответстващо на самия каталог и с име "..", съответстващо на родителския му каталог. Процесът трябва да има права x за всички каталози на пътя в пълното име и право w за родителския каталог. Грешка ще е и ако вече съществува файл с име *dirname*, независимо от типа му.

```
#include <unistd.h>
int rmdir(const char *dirname);
```
Връща 0 при успех, -1 при грешка.

Примитивът `rmdir` унищожава каталог с име *dirname*, който трябва да е празен, т.е. да съдържа само двата стандартни записа - "." и "..". Процесът трябва да има права `x` за всички каталози на пътя в пълното име и право `w` за родителския каталог. Ако за родителския каталог е вдигнат Sticky бита, то са необходими същите права както при примитива `unlink`.

4.3. Текущ каталог - `chdir` и `fchdir`

Всеки процес има свой независим текущ каталог. Този каталог се използва при търсене по относително име на файл. След вход в системата текущ каталог за процеса login-shell става началния каталог на съответния потребител, съхраняван във файла `/etc/passwd`. Текущият каталог е атрибут на процеса, а началният каталог е атрибут на потребителя. Текущият каталог, както и отворените файлове, се наследява от породените процеси, т.е. всеки процес наследява текущия каталог от своя процес-баща. След това процес може да го промени чрез примитивите `chdir` и `fchdir`.

```
#include <unistd.h>
int chdir(const char *dirname);
int fchdir(int fd);
```
Връща 0 при успех, -1 при грешка.

Указаният чрез аргумента каталог *dirname* става новия текущ каталог на процеса. Текущият каталог представлява активен файл, т.е. при изпълнение на `chdir` индексният описател на новия каталог се зарежда в таблицата на индексните описатели, а i-node на стария текущ каталог се освобождава. Процесът трябва да има права `x` за всички каталози в пълното име *dirname*. При успех връща 0, а при грешка -1 и текущият каталог не се променя. Действието на `fchdir` е аналогично, разликата се състои в начина на идентифициране на каталога, чрез име в `chdir` и файлов дескриптор в `fchdir`.

Пример

Програма 4.3 илюстрира примитива `chdir`. Тъй като текущият каталог е атрибут на процеса, който се предава само от процес-баща към процес-син, то изменението му в един процес няма ефект върху процеса-баща.

```
/* ------------------------------------------------------------ */
/* Example of chdir */

#include "ourhdr.h"

main(void)
{
  if (chdir("/tmp") < 0)
     err_sys_exit("chdir error");

  printf("chdir to /tmp succeeded\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ pwd
/home/moni/SysProg/src
$ a.out
chdir to /tmp succeeded
$ pwd
/home/moni/SysProg/src
```

Текущият каталог на процеса shell, изпълнил програмата, е непроменен. По тази причина командата `cd`, винаги се реализира като вътрешна команда на командния интерпретатор.

4.4. Четене от каталог

Точният физически формат на запис от каталог зависи от версията на файловата система в Unix или Linux. Ранните версии на Unix, които използват традиционната UNIX System V файлова система (s5fs), реализират записи с фиксиран размер от 16 байта: 2 байта за номер на i-node и 14 за името на файла. Файловите системи UFS в BSD и ext2/ext3 в Linux въвеждат дълги имена на файлове (до 255 байта) и реализират записи с променлив размер. Това означава, че програма, която отваря каталог чрез `open` и чете чрез `read` или друг примитив (напр., `getdents` в Linux), ще е системно зависима. Затова, за да се опрости четенето от каталог, се разработват библиотечни функции за четене от каталог, които вече са част от стандарта POSIX.1. Ще разгледаме основните библиотечни функции за каталози.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);
```

Връща указател при успех, `NULL` при грешка.

```
struct dirent *readdir(DIR *dir);
```

Връща указател при успех, `NULL` при EOF или грешка.

```
void rewinddir(DIR *dir);

int closedir(DIR *dir);
```

Връща 0 при успех, -1 при грешка.

Структурата `dirent` е определена в заглавния файл `<dirent.h>` и е системно зависима. Но съдържа поне следните два елемента:

```
struct dirent {
   ino_t d_ino;              /* inode number */
   char d_name[NAME_MAX+1];  /* file name (null-terminated) */
};
```

Елементът `d_name` съдържа собственото име на файла, като в края на низа е добавен символа `'\0'`. Значението на `NAME_MAX` зависи от типа на файловата система и може да бъде определено чрез функцията `pathconf` (виж Програма 1.3). Елементът `d_ino` съдържа номера на i-node на файла.

Структурата `DIR` се свързва с отворен каталог и се използва от останалите функции (аналог е на структурата `FILE`, използвана от функциите за поточен вход/изход от файлове). Функцията `opendir` отваря каталог *`dirname`* за четене и връща указател към структура `DIR`. Този указател се използва при другите три функции като идентификатор на отворения каталог и се задава чрез аргумента *`dir`*. При грешка функцията `opendir` връща указател `NULL`.

Функцията `readdir` чете поредния следващ запис от каталога (при първо извикване чете първия запис), като наредбата на записите не е по името на файла и е системно зависима. Прочетеният запис се предава чрез структура `dirent`, указател към която се връща като значение на функцията при успех. При EOF (достигане край на каталога) или при грешка функцията връща указател `NULL`. Ако след `opendir` многократното извикваме `readdir` докато върне `NULL`, ще прочетем последователно записите в каталога.

Функцията `rewinddir` позиционира в началото на каталога. Това позволява след отваряне на каталог да се изпълнят няколко последователни преминавания през него.

Функцията `closedir` затваря каталога и освобождава идентификатора му *`dir`*. Освен описаните тук функции, се реализират и други функции за поточен вход от каталог.

Пример

Програма 4.4 приема един аргумент, който е име на каталог. Извежда имената на файловете в каталога. Тази програма е примитивен вариант на командата `ls` без опции.

```
/* ------------------------------------------------------------ */
/* List files in directory */

#include <sys/stat.h>
#include <dirent.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  struct dirent *dep;
  DIR *dp;

  if (argc < 2)
      err_exit("usage:a.out directory");

  dp = opendir(argv[1]);
  if (dp == NULL)
      err_sys_exit("open error for %s", argv[1]);

  while ((dep=readdir(dp)) != NULL) {
     if ( dep->d_name[0] == '.' ) continue;
     printf("%s\n",dep->d_name);
  }
  closedir(dp);
  exit(0);
}
```

Ако изпълним програмата ще получим изход от вида:

```
$ ls dir1
dd     file1  out
$ a.out dir1
dd
file1
out
$ a.out dir1/out
open error for dir1/out: Not a directory
$ a.out dir1/dir2
open error for dir1/dir2: No such file or directory
```

Пример

Програма 4.5 приема един аргумент, който е име на каталог. Извежда справка за съдържанието на каталога, която включва имената и размера на обикновените файлове.

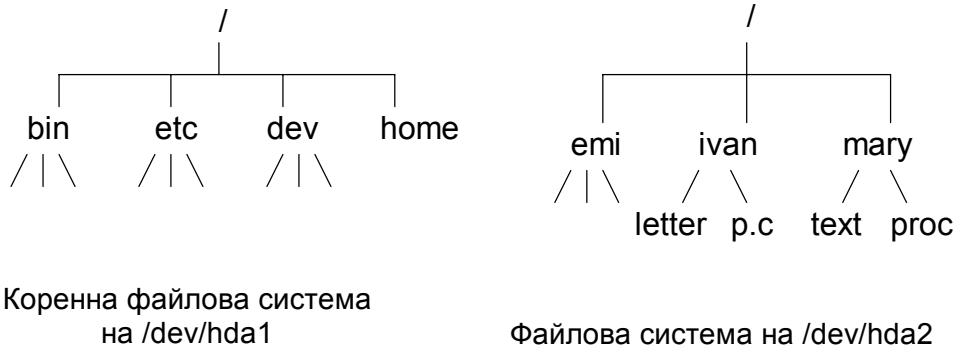
```
/* ------------------------------------------------------------ */
/* List regular files in a directory */

#include <sys/stat.h>
#include <dirent.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  struct dirent *dep;
  struct stat sbuf;
  DIR *dp;
  char name[4096];

  if (argc < 2) {
      err_exit("usage: a.out dirname");

  dp = opendir(argv[1]);
  if (dp == NULL)
      err_sys_exit("open error for %s", argv[1]);

  if ( chdir(argv[1]) < 0 )
    err_sys_exit("chdir error");

  while ((dep = readdir(dp)) != NULL) {
    if (dep->d_name[0] == '.') continue;

    if(stat(dep->d_name, &sbuf) ==-1) {
      err_sys("stat error for %s", dep->d_name);
      continue; }

    if( S_ISREG(sbuf.st_mode) )
      printf("%s: %d\n", dep->d_name, sbuf.st_size);
  }
  closedir(dp);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ ls -l dir1
total 28
drwxr-xr-x  2 moni staff 4096 Sep  7 13:55 dd
-rw-r-----  1 moni staff 8204 Sep  7 13:04 file1
-rw-r--r--  1 moni staff  543 Sep  7 13:07 out
$ a.out dir1
out: 543
file1: 8204
```

4.5. Монтиране и демонтиране на файлова система и специални файлове

Системните примитиви `mount` и `umount` позволяват за потребителя винаги да се изгражда единна йерархична файлова система, независимо от броя на носителите (флопи дискове, твърди дискове или техни дялове). На всеки носител е изградена йерархична файлова система чрез командата `mkfs`, една от които съдържа програмата за начално зареждане и ядрото на операционната система, и се нарича коренна файлова система. Чрез примитива `mount` некоренна файловата система на определен носител може да бъде присъединена (монтирана) към коренната файлова система. Например, нека конфигурацията включва две дискови устройства и върху тях са изградени файловите системи, показани на Фиг.4.2.

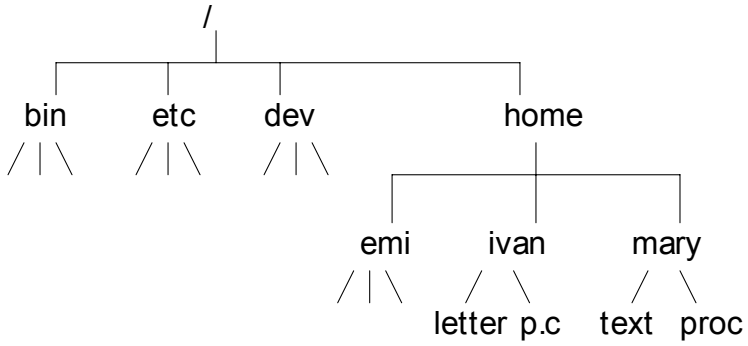


Фиг. 4.2. Две файлови системи преди монтирането

След изпълнение на системния примитив

```
mount("/dev/hda2", "/home", . . . );
```

за потребителя файловата система ще има следната структурата, показана на Фиг.4.3.



Фиг. 4.3 Файловата система след монтирането на `/dev/hda2`

Следователно файловете, които са разположени на устройството `/dev/hda2`, ще са достъпни за потребителя чрез пълно име, напр., `/home/ivan/letter`.

Разрушаване на връзката между коренната и коя да е друга монтирана файлова система се извършва чрез системния примитив `umount`. Точният синтаксис на двата примитива е системно зависим и в Linux например е следния:

```
#include <sys/mount.h>

int mount(const char *special, const char *dirname,
    const char *fstype, unsigned long flags, const void *data);

int umount(const char *dirname);
```

Връща 0 при успех, -1 при грешка.

Аргументът *special* е име на специалния файл на монтирана файлова система. При изпълнение на `mount` каталогът *dirname*, в който се монтира трябва да съществува, в него вече да не е монтирана файлова система и да не е активен в момента на монтиране (напр., да не е нечий текущ каталог). Аналогично, изпълнението на `umount` завършва с грешка и не се демонтира файловата система, ако в момента там има активен файл (отворен обикновен файл или текущ каталог). Останалите параметри са специфични за Linux. Процесът, в който се изпълнява `mount` или `umount` трябва да принадлежи на привилегирования потребител (да е с `euid` 0). При успех и двата примитива връщат 0, а при грешка -1.

В структурата `stat` има два елемента `st_dev` и `st_rdev`, които съдържат номер на устройство. Каква е разликата между тях?

- Всяко устройство се идентифицира в системата с два номера: номер на тип (major device number) и номер на устройството в типа (minor device number). Тези два номера заедно ще наричаме номер на устройство и се съхраняват в примитивния тип `dev_t`.
- За достъп до двете части от номера на устройството има макроси - `major` и `minor`.
- За всеки файл елементът `st_dev` съдържа номера на устройството, където се съхранява индексния описател и данните на файла.
- Само специалните файлове имат значение в елемента `st_rdev`. Значението е номер на устройство, на което съответства специалния файл.

Пример

Програма 4.6 приема произволен брой аргументи, които са имена на файлове. За всеки файл извежда номера на устройството, където се съхранява файла, а за специалните файлове извежда и номера на устройството, което му съответства.

```
/* ------------------------------------------------------------ */
/* Print st_dev and st_rdev values */

#include <sys/stat.h>
#include <sys/sysmacros.h>
#include "ourhdr.h"

main(int argc, char *argv[])
{
  int i;
  struct stat sbuf;
  for (i = 1; i < argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &sbuf) < 0) {
       err_sys("lstat error");
       continue;
    }
    printf("dev = %d/%d", major(sbuf.st_dev), minor(sbuf.st_dev));

    if (S_ISCHR(sbuf.st_mode)) {
      printf(" (char) rdev = %d/%d", major(sbuf.st_rdev),
                                     minor(sbuf.st_rdev));
    }
    if (S_ISBLK(sbuf.st_mode)) {
      printf(" (block) rdev = %d/%d", major(sbuf.st_rdev),
                                      minor(sbuf.st_rdev));
    }
    printf("\n");
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ a.out / /opt /dev/tty1 /dev/hda1 /dev/hdb2
/: dev = 3/1
/opt: dev = 3/66
/dev/tty1: dev = 3/1 (char) rdev = 4/1
/dev/hda1: dev = 3/1 (block) rdev = 3/1
/dev/hdb2: dev = 3/1 (block) rdev = 3/66
$ mount
/dev/hda1 on / type ext2 (rw)
none on /proc type proc (rw)
```

```
/dev/hda2 on /dosc type vfat (rw)
/dev/hdb2 on /opt type ext2 (rw)
/dev/hdb1 on /sec type ext2 (rw)
$ ls -l /dev/tty1 /dev/hda1 /dev/hdb2
crw-------   1 root     tty        4,   1 Oct 11 11:59 /dev/tty1
brw-rw----   1 root     disk       3,   1 May  5  1998 /dev/hda1
brw-rw----   1 root     disk       3,  66 May  5  1998 /dev/hdb2
```

Първите два аргумента `/` и `/opt` са имена на каталози, а следващите три са имена на специални файлове. Коренният каталог и каталогът `/opt` имат различни номера на устройства, защото се намират на различни файлови системи. Това се вижда от изхода на командата `mount`. След това чрез изхода от командата `ls` проверяваме типа и номера на устройство за специалните файлове `/dev/tty1`, `/dev/hda1` и `/dev/hdb2`.

4.6. Системни примитиви `sync` и `fsync`

Ядрото изпълнява входно-изходните операции за блоковите специални файлове като използва буфериране (определен брой буфери, намиращи се в пространството на ядрото и наричани буферен кеш). Когато се изпълнява примитив `read` първо се чете дисковия блок в системен буфер и след това исканите байтове се копират в областта на процеса. Аналогично при изпълнение на примитив `write`, даните първо се копират в съответния буфер, той се отбелязва за запис и примитивът завършва, но действителното записване на буфера на диска се отлага за по-късен момент. Това е така наречената стратегия на отложен запис (delayed write). Обикновено ядрото записва отбелязаните блокове когато трябва да освободи буфер за друга входно-изходна операция. Ако е вдигнат флаг `O_SYNC` при `open`, то `write` връща управлението след физическото записване на данните и управляващата информация на диска, т.е. писането на диска е синхронно.

Обикновено за повечето файлове се прилага стратегия на отложен запис. Ако искаме да синхронизираме данните на диска със съдържанието на буферния кеш, то можем да използваме примитивите `sync` и `fsync`.

```
#include <unistd.h>
void sync(void);
int fsync (int fd);
```

Връща 0 при успех, -1 при грешка.

Примитивът `sync` планира запис на диска за всички отбелязани буфери в буферния кеш, но не чака действителното приключване на записа. Обикновено `sync` се извиква периодично на всеки 30 секунди от специален процес-демон. Това гарантира регулярното синхронизиране на диска с буферния кеш. Има и команда `sync`, която всъщност извиква примитива.

Примитивът `fsync` приема аргумент *`fd`*, който е файлов дескриптор на отворен файл и синхронизира данните само за него. Той изчаква да завърши дисковата операция и тогава връща управлението. Каква е разликата между използването на флаг `O_SYNC` при отваряне на файл и примитива `fsync`? При флаг `O_SYNC` запис на диска се извършва при всяко извикване на `write`, а при `fsync` само когато се извика примитива. Ако искаме да сме сигурни, че всички изменения във файла са записани на диска, трябва да извикваме `fsync` преди `close`. Защото ако не използваме `fsync`, то `close` може да завърши успешно, но данните да са още в буферния кеш. По късно може да се случи грешка при опит на ядрото да запише данните на диска, но тогава програмата не ще може да реагира.