СИСТЕМНО ПРОГРАМИРАНЕ

записки на лекциите

Специалност Компютърни науки

гл. ас. д-р Моника Филипова

Катедра Изчислителни системи, ФМИ

зимен семестър на уч. 2013/2014г.

СЪДЪРЖАНИЕ

Първа глава

Първа глава

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. Ще започнем с кратък преглед на основните понятия, които по-нататък ще бъдат по-подробно разгледани от гледната точка на програмиста.

1.1. Файлове и каталози

Повечето операционни системи (ОС) реализират йерархична (дървовидна) организация на файловете чрез специален тип файл, наречен каталог (directory). Всичко започва от коренния каталог, чието име е символът „/”. Логически всеки каталог съдържа записи за файлове и други каталози. Освен обикновените файлове и каталозите, много ОС реализират и други типове файлове: специални файлове (character special device files, block special device files), програмни канали (FIFO files), символни връзки (symbolic links) и др.

Всеки файл има **собствено име**, което се съдържа в записа от родителския му каталог. В някои Unix системи собственото име е ограничено до 14 символа, в по-съвременните весии на Unix, в Linux и в POSIX се въвеждат дълги собствени имена - до 255 символа.

Всеки файл има едно **абсолютно пълно име** (absolute path name), което съответства на единствения път в дървото от коренния каталог до файла. То представлява списък от имената на каталозите в пътя, разделени със символа „/”, започвайки с името на коренния каталог - /.

Всеки процес има свой **текущ каталог** (current directory, работен каталог, working directory). Относно този каталог започва **относителното пълно име** (relative path name) на файл. То представлява списък от имената на каталозите в пътя, тръгвайки от текущия в момента каталог (отново разделени със символа „/”).

Всеки потребител, регистриран в системата има свой **начален каталог** (home directory), чието име се помни във файла `/etc/passwd`. Веднага след вход в системата, текущ каталог е началния каталог на съответния потребител.

1.2. Програми и процеси

Програма е файл, съдържащ изпълним код и съхраняван на диска като обикновен файл. След като програмата бъде заредена в паметта и започне нейното изпълнение под управлението на ядрото, започва животът на процеса. Процес е програма в хода на нейното изпълнение. В някои ОС се използва и термина задача (task).

Всеки процес има уникален идентификатор, който се нарича **process ID** (pid) Идентификаторът на процес е неотрицателно цяло число и ОС осигурява неговата неизменност през целия живот на процеса.

ОС осигурява четири основни системни примитива за управление на процесите - `fork`, `exec`, `exit` и `wait`. Нов процес се създава само чрез `fork`. Процесът, извикващ `fork`, се нарича процес-баща, а новосъздаденият е процес-син. Чрез `exec` процес може да извика за изпълнение нова програма. Процес завършва своя живот когато изпълни `exit`. Чрез `wait` процес-баща може да изчака завършването на свой процес-син.

1.3. Потребителски идентификатори

Всеки потребител, регистриран в системата има потребителско име (което задава при вход в системата) и уникален числов идентификатор, наричан потребителски идентификатор (user ID или uid). Този uid е неотрицателно цяло число и се използва в структурите на ОС при определяне правата на потребителите. Потребителят с uid 0 е администратора или привилегирования потребител, известен още като `root`.

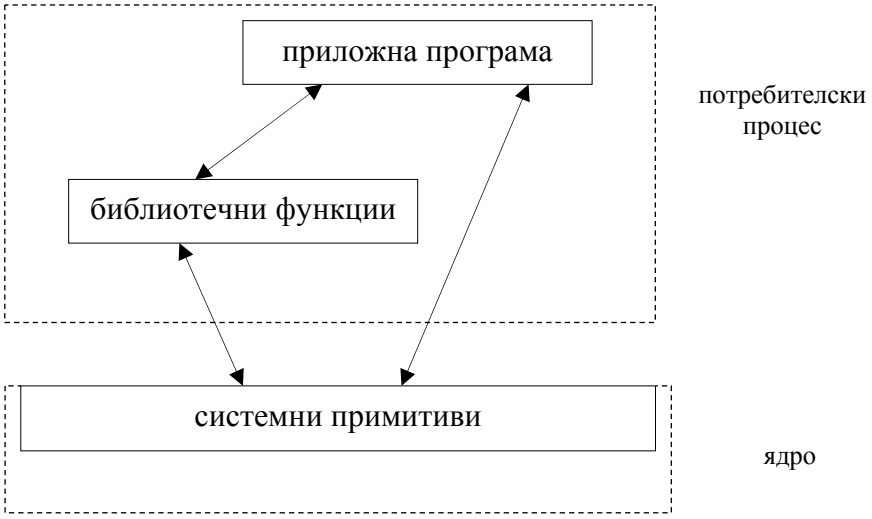
Всеки потребител принадлежи на определена потребителска група, която се определя от администратора и се идентифицира чрез идентификатор на група (group ID или gid). Идентификаторът на група е също уникално, неотрицателно цяло число.

С всеки процес се свързва uid и gid на потребителя, който го е стартирал. (Всъщност всеки процес има още една двойка uid и gid, наричани ефективни, но за това по-късно.)

1.4. Системни примитиви и библиотечни функции

Системните примитиви са програмният интерфейс на ядрото на ОС. В ранните версии на UNIX 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 C. 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. Те са входове в ядрото и затова ги наричат системни извиквания. Системните примитиви обикновено са документирани в раздел 2 от документацията.

Библиотеките на езика C включват още много други функции, които не са директни входове в ядрото, но някои могат да съдържат обръщение към системни примитиви. Напр., функцията `atoi` се реализира изцяло от код 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

1.5. Обработка на грешки

При успех функцията на системния примитив обикновено връща неотрицателно цяло число. Когато се случи грешка при изпълнение на системен примитив, функцията връща отрицателно цяло 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 значението в `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 значение `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 <stdlib.h>
#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 <stdlib.h>
#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`). Всички те извеждат съобщението на стандартгия изход за грешки.

`void err_sys_exit(char *frm, …);`

Извежда съобщение след грешка в системен примитив и завършва процеса.

`void err_sys_ret(char *frm, …);`

Извежда съобщение 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`.

1.6. 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 Linux Kernel 2.6, а в някои случаи и Linux Kernel 2.0.

Съществуват различни вълшебни константи, които определят различни ограничения на реализацията. Ограниченията се разделят 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 могат да се поверят чрез библиотечните функции `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("OPEN_MAX           =", _SC_OPEN_MAX);
  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("_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 Linux Kernel 2.0 ------*/

```
ARG_MAX            = 131072
CHILD_MAX          = 999
OPEN_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
_POSIX_NO_TRUNC    = 1
_POSIX_CHOWN_RESTRICTED = 1
```

/*------ в Linux Kernel 2.6 ------*/

```
ARG_MAX            = 131072
CHILD_MAX          = 999
OPEN_MAX           = 1024
_POSIX_JOB_CONTROL = 1
_POSIX_SAVED_IDS   = 1
_POSIX_VERSION     = 200112
LINK_MAX           = 32000
NAME_MAX           = 255
PATH_MAX           = 4096
PIPE_BUF           = 4096
_POSIX_NO_TRUNC    = 1
_POSIX_CHOWN_RESTRICTED = 1
```

1.7. 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

Втора глава

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 (с всеки файлов дескриптор) се свързва и **режим на отваряне** на файла, който определя начина на достъп до файла 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

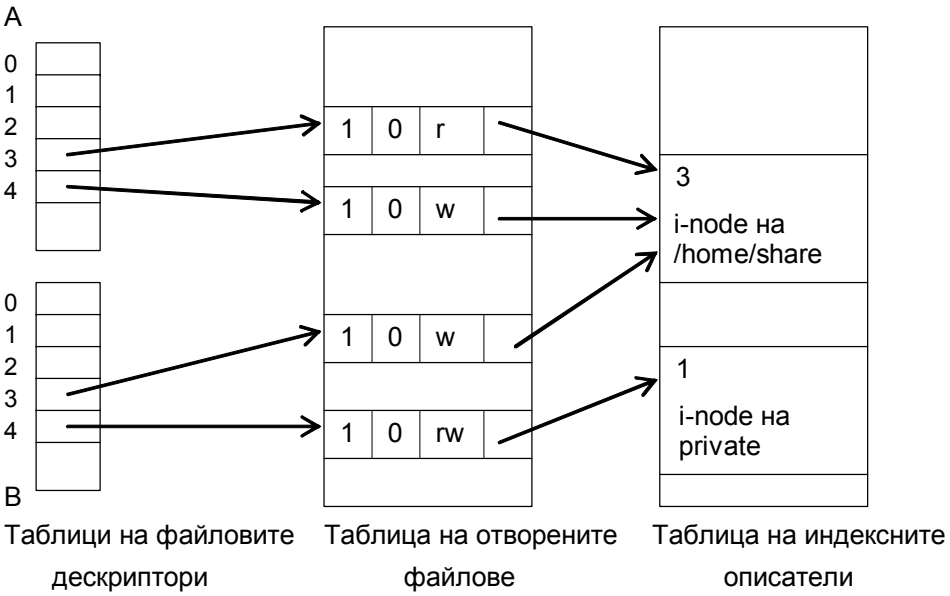
   При всяко отваряне на файл се отделя един запис в тази таблица, който съдържа:
   - текущата позиция във файла
   - режима на отваряне на файла
   - указател към съответния запис от таблицата на индексните описатели
   - брояч (брой указатели, насочени към записа, 1 след отваряне).

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.

Фиг. 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. В този случай аргументът *mode* определя кода на защита на новосъздадения файл. Функциите на системния примитив се конкретизират чрез аргумента *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.

Всъщност 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 новосъздаденият 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 успешно:

```
fdr = open("/home/share", O_RDONLY);
fdw = open("/home/share", O_WRONLY);
```

а процес В изпълни също успешно 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`

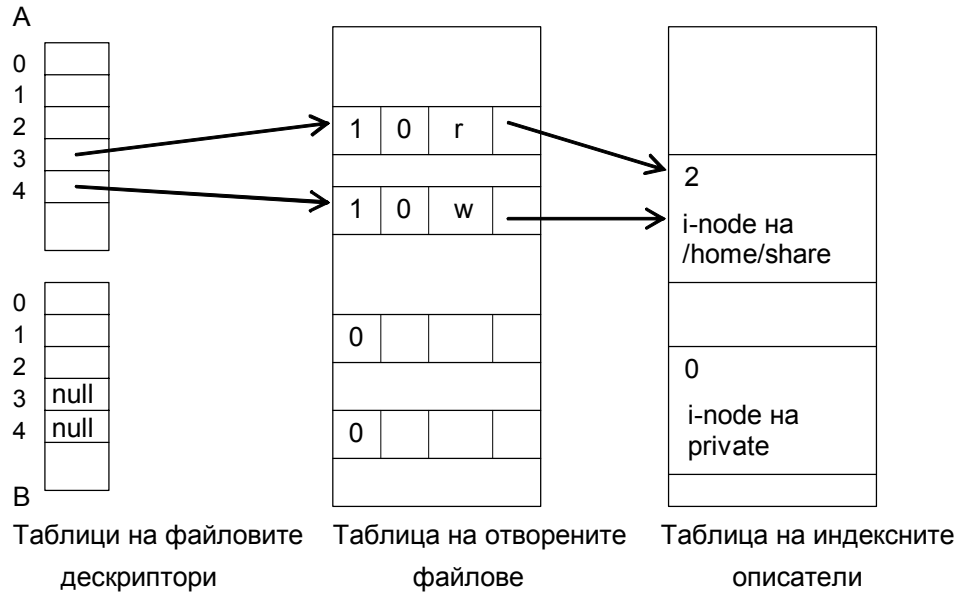
Примитивът `close` затваря отворен файл, т.е. прекратява връзката между процеса и файла, като освобождава файловия дескриптор, който след това може да бъде използван при последващи изпълнения на `creat` или `open` от процеса.

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

Връща 0 при успех, -1 при грешка.

Фиг.2.2 изобразява вида на таблиците след като процес B изпълни:

```
close(fd1);
close(fd2);
```



Фиг. 2.2. Системните таблици след `close`

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 аргумента *`buffer`*. Указателят на текуща позиция се увеличава с действителния брой прочетени байта, т.е.

сочи байта след последния прочетен байт. Функцията 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 значението в аргумента *nbytes* :

- При четене от обикновен файл, ако до края на файла има по-малко от *nbytes* байта. Ще прочете толкова байта, колкото има до края на файла. Следващият `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-нататъшно 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, т.е. е 0 (дори ако 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 ще се промени текущата позиция, 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 определени 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 = open("file.hole", O_WRONLY|O_CREAT|O_TRUNC, 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 символ `'\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 успешно 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:

```
fd = open("/home/share", O_WRONLY);

if (lseek(fd, 0, 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 A	**Процес B**
`lseek`	
	`lseek`
	`write`
`write`	

При такава последователност процесът A пише върху данните записани от процеса B.

Проблемът тук се състои в това, че логическата операция „позициониране в края на файла и писане" се реализира чрез два системни примитива и не е атомарна. 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`. Следователно, друг вариант за решение е:

```
fd = open("/home/share", O_WRONLY|O_APPEND);

if (write(fd, buf, 10) != 10)       /* position to EOF and write */
    err_sys_exit("write error");
```

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);
```

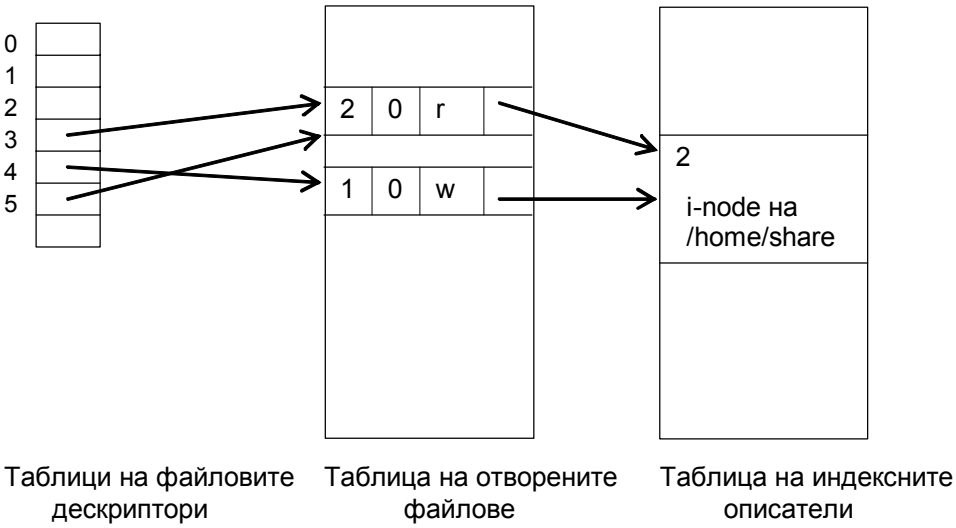
Връща нов файлов дескриптор при успех, -1 при грешка.

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 изпълнил 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.2.3* 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 2.3. 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 обработчикът да промени файловите дескриптори.

2.8. Системен примитив `fcntl`

С този примитив могат да се четат или изменят различни свойства на отворен файлов дескриптор.

```
#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`:

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

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

`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`.

`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.

Пример

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

```
/* ------------------------------------------------------------ */
/* Print 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. Процесът получава тези файлове отворени и не знае името на файла, което се крие зад съответния файлов дескриптор. Например, нека в Програма 2.1 искаме да използваме синхронно писане при копирането на файла, т.е. всеки `write` да чака докато данните бъдат записани на диска преди да върне управлението. В началото преди цикъла `while` 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");
```

Трета глава

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 могат да 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. Също така `st_blksize` съдържа препоръчвания размер на блок при четене и писане във файла.

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 – тип на файла и кода му на защита. 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) е файл, съдържащ данни в някакъв формат.
2. Каталог (directory). Това е файл, съдържащ записи за други файлове, всеки от които съдържа 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, напр., терминал, печатащо устройство 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). Това е тип файл, реализиращ механизъм за междупроцесни комуникации.
7. Сокет (socket) е 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;
  struct stat statbuf;
  char *ptr;

  for (i=1; i<argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &statbuf) < 0) {
       err_sys_ret("stat error");
       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/hda1 /dev/null xx
a.out: regular
/etc: directory
/dev/tty: character special
/dev/hda1: block special
/dev/null: character special
xx: stat error: No such file or directory
```

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 условията в операторите `if` 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, а не имената им.) 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 файл от тип каталог:

- Право `r` означава правото да четем съдържанието на каталога. Напр., 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. Напр., за изпълнение на командата `rm dir/text` се изисква право `w` за каталога `dir`.
- Право `x` означава търсене на файлове в каталога и позициониране в каталог. Напр., 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 може да 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.

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 и Linux 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.

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.
- За да получим информация за файл чрез примитив `stat` е необходимо да имаме право `x` за всички каталози по пътя към файла.

Следователно, правата на достъп до каталозите в пълното име на файла се използват при проверка на правата за достъп до файл. (Това се нарича traversal permissions.)

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.

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 примитива `umask` се зарежда маската за създаване на файл за процеса. Аргументът *`cmask`* се конструира като побитово или на деветте константи от *Фиг.3.2*. Новата маска на процеса е *`cmask`*`&0777`, т.е. от значение са младшите 9 бита на *`cmask`*. Това е един от малкото примитиви, който не връща грешка. Системният примитив връща старото значение на маската, което може да бъде съхранено и по-късно възстановено.

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 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. 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`*. Забележете, че няма примитив `lchmod`, тъй като кодът на защита на файл от тип символна връзка не се използва.

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 функцията 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/или 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, ако *`filename`* е символна връзка.

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 функцията 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 и символна връзка. 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 символни връзки значението е брой байтове в името на файла, към който сочи връзката, например:

```
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`. Елементът `st_blksize` съдържа препоръчвания размер на блок при четене и писане във файла. Елементът `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` и извежда размера в брой KB. 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
```

Размерът на блок във файловата система, където е тестван този пример, е 1024 байта. Какъв ще е резултатът ако блокът е с размер 4096 байта?

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 и повече.

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

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 във функцията, е определена в заглавния файл `<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 (`root`).

Като аргумент на функцията не се задава значение за времето на последно изменение на i-node на файла, защото то автоматично се изменя когато се изпълнява примитива `utime`.

Пример

Програма 3.5 приема аргументи, които 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.

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

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

Четвърта глава

ИЗГРАЖДАНЕ НА СТРУКТУРАТА НА ФАЙЛОВА 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 в един том 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, т.е. осигурява връзка между името и данните и атрибутите на файла.
- 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.

```
fd = open(temporary, O_RDWR|O_CREAT|O_TRUNC, mode);
unlink(temporary);
   /* четене и 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_ret("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_ret("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
$ ls -l file1
ls: file1: No such file or 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 *`fromname`*.

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 на реда) и сочи към файл `/no/such/file` (името след символите ->).

Тъй като примитивът `open` следва символната връзка, то ако искаме да отворим самия файл символна връзка и да прочетем съдържанието му, трябва да използваме примитива `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`* (не добавя символа '\0' на края) 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 аргументът е име на символна връзка). *Фиг.4.1* обобщава тази информация за основните системни примитиви, които разглеждаме в глави 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);
```

Връщат 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 функцията 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 използват 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` или `readdir` в различни версии на Linux), ще е системно зависима. Затова, за да се опрости четенето от каталог, се разработват библиотечни функции за поточен вход от каталог, които вече са част от стандарта POSIX.1. Ще разгледаме основните библиотечни функции за четене от каталог.

```
#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 `DIR` (нарича се directory stream) се свързва с отворен за четене каталог и се използва от останалите функции (аналог е на структурата `FILE`, използвана от функциите за поточен вход/изход от обикновен файл).

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.

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 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 file names in a directory */

#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
out
file1
$ 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)
```

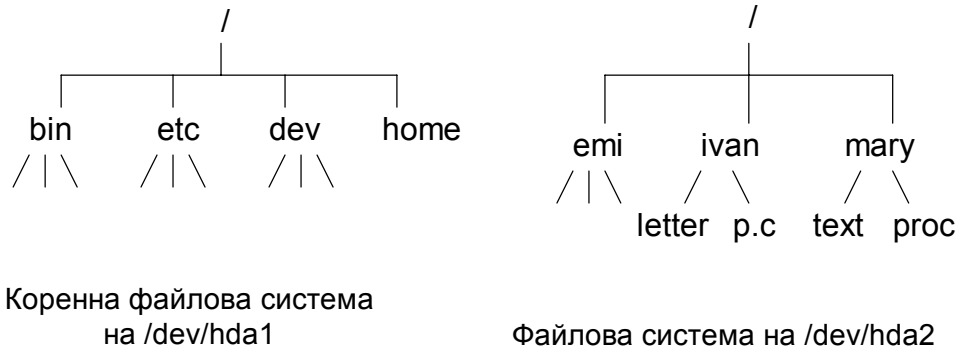
```
      err_sys_exit("open error for %s", argv[1]);

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

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

    if(lstat(dep->d_name, &sbuf) == -1) {
      err_sys_ret("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 32
drwxr-xr-x  2 moni staff 4096 Nov  4 12:41 dd
-rw-r--r--  1 moni staff 4714 Nov  4 13:05 file1
-rw-r--r--  1 moni staff  874 Nov  4 12:41 out
lrwxrwxrwx  1 moni staff    5 Nov  4 12:50 xx_link -> ../xx
$ a.out dir1
out: 874
file1: 4714
```

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 (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*.

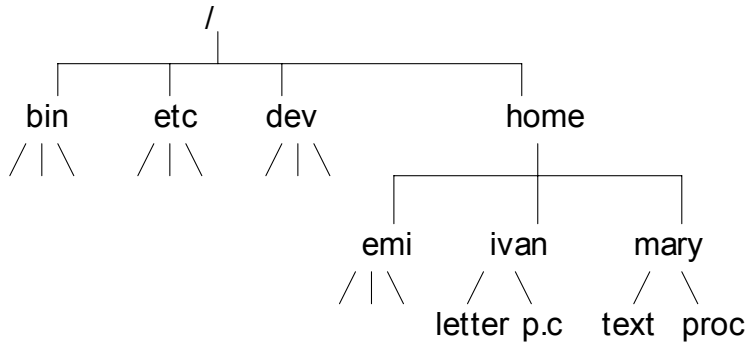


Фиг. 4.2. Две файлови системи преди монтирането

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);
```

Връщат 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 аргументи 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 системен 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_ret("stat 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. Буфериране на входно/изходните операции - системни примитиви `sync`

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

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

Връщат 0 при успех, -1 при грешка.

Примитивът `sync` планира запис на диска за всички отбелязани буфери в буферния кеш, но може да не чака действителното приключване на записа. (В някои версии на Linux се чака приключването 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 и атрибутите само за него. Примитивът `fdatasync` синхронизира само данните на файла, без атрибутите му. И двата примитива изчакват да завърши дисковата операция и тогава връщат управлението. (В някои случаи, в зависимост от типа на дисковото устройство, данните може и да не са стигнали до дисковата памет при връщане от примитива.)

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.

Пета глава

УПРАВЛЕНИЕ НА ПРОЦЕСИ

Тази глава е посветена на абстракцията процес и основните системни примитиви за работа с процеси. Това включва създаване на процес, изпълнение на програма и завършване на процес. Разглеждат се атрибутите на процес и системните примитиви, чрез които се изменят значенията на някои от атрибутите на процес. Всички процеси са организирани в йерархична структура, отразяваща тяхното пораждане. Освен тази основна връзка между процесите „баща-син", съществуват и други, свързани с понятията група процеси и сесия.

5.1. Контекст на процес

Преди да започнем разглеждането на системните примитиви за управление на процеси, да видим какво представлява обкръжението на един отделен процес. Ще видим как се извиква функцията `main`, как се предават аргументи от командния ред към програмата, как процес може да използва променливите от обкръжението си и ще си припомним някои основни атрибути на процес.

Процес е програма в хода на нейното изпълнение**.** Програма на езика C започва изпълнението си от функция `main`, чийто прототип е:

```
int main(int argc, char *argv[]);
```

където `argc` е броят на аргументите, предадени от командния ред, а `argv` е масив от указатели към самите аргументи. Ядрото стартира изпълнението на нова програма при извикване на системния примитив `exec`. Тогава чрез аргументите на примитива `exec` могат да се предадат аргументи от командния ред към новата програма. Тази възможност се използва от програмата shell (и затова ги наричаме аргументи от командния ред или command line arguments) и също така беше използвана в много от примерите до сега.

Пример

Програма 5.1 извежда на стандартния изход всички аргументи от командния ред, т.е. има действие подобно на командата `echo`.

```
/* ------------------------------------------------------------ */
/* Echo all command line arguments */

#include "ourhdr.h"

main(int argc, char *argv[])
{
  int  i;

  for (i=0; i<argc; i++)
       printf("arg[%d]: %s\n", i, argv[i]);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ a.out argument1 xxxx third
arg[0]: a.out
arg[1]: argument1
arg[2]: xxxx
arg[3]: third
```

На една програма, освен аргументи, може да се предава и списък от променливи на обкръжението (environment variables). Списъкът от променливи се предава като масив от указатели на символни низове, който завършва с указател `NULL`. Всеки указател сочи към символен низ, който има вида:

name=value

където *name* е името на променливата, а *value* е значението й. Програмата може да работи с променливите чрез глобалната променлива `environ`, която съдържа адреса на масива от указатели.

```
extern char **environ;
```

Повечето Unix системи поддържат още един трети аргумент на функцията `main`, който е адрес на обкръжението:

```
int main(int argc, char *argv[] char *envp[]);
```

Според POSIX стандарта трябва да се използва променливата `environ` вместо третия аргумент на `main`. За да обходим всички променливи на обкръжението, трябва да използваме променливата `environ`.

Пример

Програма 5.2 извежда на стандартния изход всички аргументи от командния ред и всички променливи от обкръжението и техните значения, т.е. има действие подобно на командата `echo`, последвана от командата `set`, извикана без аргументи.

```
/* ------------------------------------------------------------ */
/* Echo all command line arguments and environment strings */

#include "ourhdr.h"

main(int argc, char *argv[])
{
  int  i;
  char **ptr;
  extern char **environ;

  for (i=0; i<argc; i++)            /* echo command line arguments */
       printf("arg[%d]: %s\n", i, argv[i]);

  for (ptr = environ; *ptr != 0; ptr++) /* echo environment strings */
       printf("%s\n", *ptr);
  printf("\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида:

```
$ a.out
arg[0]: a.out
HOME=/home/moni
SHELL=/bin/bash
PS1=[\u@\h \W]\$
USER=moni                  <и още много други променливи, които не са показани>
```

В стандартите ANSI C и POSIX, а и в библиотеките на различните Unix и Linux системи са включени функции за достъп до определена променлива на обкръжението: `getenv(3)`, `putenv(3)`, `setenv(3)` и `unsetenv(3)`.

```
#include <stdlib.h>
char *getenv(const char *name);
                                    Връща указател към value, NULL при грешка.
int putenv(const char *str);
int setenv(const char *name, const char *value, int rewrite);
                                    Връщат 0 при успех, -1 при грешка.
void unsetenv(const char *name);
```

За извличане значението на променлива по името й *name* се използва функцията `getenv`, която връща указател към низа *value*. За да се добави нова променлива или да се измени значението на съществуваща променлива от обкръжението, могат да се използват функциите `putenv` и `setenv`. Аргументът на `putenv` съдържа низа *name*=*value*. Ако в обкръжението не съществува променлива *name*, то се добавя, а ако съществува значението й се променя на *value*. При функцията `setenv` изменението на значението на променливата, ако съществува, зависи от аргумента *rewrite*. Чрез функцията `unsetenv` променлива се изключва от обкръжението по името й *name*.

Пример

Програма 5.3 извежда на стандартния изход значението на конкретна променлива от обкръжението, добавя нова променлива, изменя значението й, и накрая я изключва, т.е. има действие подобно на командите: `echo $HOME; X=123; unset X`.

```
/* ------------------------------------------------------------ */
/* Echo, set and unset environment variables */

#include "ourhdr.h"

main(int argc, char *argv[])
{
  char name[]="HOME";
  char name2[]="X";
  char value[256];
  char str[]="X=123";
  char *ptr;

  ptr = value;
  ptr = getenv(name);
  if (ptr == NULL)
     err_ret("No variable HOME");
  else
     printf("HOME: %s\n", ptr);

  if (putenv(str) == 0) {
     ptr = getenv(name2);
     if (ptr == NULL)
        err_ret("No variable X");
     else
        printf("X: %s\n", ptr);
  }

  setenv(name2, "aaa", 1);
  ptr = getenv(name2);
  if (ptr == NULL)
     err_ret("No variable X");
```

```
  else
     printf("After setenv X: %s\n", ptr);

  unsetenv(name2);
  ptr = getenv(name2);
  if (ptr == NULL)
     err_ret("No variable X");
  else
     printf("X: %s\n", ptr);

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

Ако изпълним програмата ще получим изход от вида:

```
$ a.out
HOME: /home/moni
X: 123
After setenv X: aaa
No variable X
```

В понятието процес освен програмата се включва и информация за процеса, която ще наричаме атрибути на процес. Да си припомним основните атрибути на процес, които се съхраняват в системните структури Таблицата на процесите и Потребителска област (user area или U area):

- идентификатор на процеса (pid)
- идентификатор на процеса-баща (ppid от parent pid)
- идентификатор на група процеси (pgid)
- идентификатор на сесия (sid)
- реален потребителски идентификатор (ruid)
- ефективен потребителски идентификатор (euid)
- реален идентификатор на потребителска група (rgid)
- ефективен идентификатор на потребителска група (egid)
- файлови дескриптори на отворените файлове
- текущ каталог
- управляващ терминал
- маска, използвана при създаване на файлове и заредена с примитива `umask`
- реакция на процеса при получаване на различни сигнали
- времена за изпълнение на процеса в системна и потребителска фаза (system and user CPU time)

Съществуват системни примитиви, които връщат значенията на различни атрибути на процеса, който ги извиква. Следва прототипа на функциите за част от тях (останалите ще разгледаме в следващите раздели). Тези функции винаги завършват успешно.

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

pid_t getpid(void);                    Връща pid на процеса.

pid_t getppid(void);                   Връща pid на процеса-баща.
```

5.2. Системен примитив `fork`

Единственият начин за създаване на нов процес от ядрото е когато съществуващ процес извика примитива `fork`. Новосъздаденият процес се нарича процес-син, а процесът, извикал `fork`, е процес-баща.

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

pid_t fork(void);
```

Връща 0 в процеса-син, pid на сина в процеса-баща, -1 при грешка.

Сложността при този примитив се състои в това, че един процес "влиза" във функцията `fork`, а два процеса "излизат" от `fork` с различни връщани значения: в процеса-баща функцията връща pid на процеса-син при успех или -1 при грешка, а в сина връща 0. Новият процес представлява почти точно копие на процеса-баща. Той изпълнява програмата на бащата и дори от мястото, до което е стигнал той, т.е. процесът-син започва изпълнението на потребителската програма от оператора след `fork`.

Пример

Програма 5.4 демонстрира създаването на нов процес и връзката баща-син между тях.

```
/* ------------------------------------------------------------ */
/* Example of fork */

#include "ourhdr.h"

main(void)
{
  pid_t pid;

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)          /* in child */
    printf("PID %d: Child started, parent is %d.\n",
    getpid(),   /* Child PID */
    getppid()); /* Parent PID */
  else {                      /* in parent */
    printf("PID %d: Started child PID %d.\n",
    getpid(),  /* Current parent PID */
    pid);      /* Child's PID */
    sleep(3);  /* Wait 3 seconds  */
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

След `fork` и двата процеса работят асинхронно и не можем да разчитаме кой ще работи първи и кой втори. Затова процесът-баща извиква библиотечната функция `sleep` (която блокира процеса за 3 секунди), за да даде възможност на сина да го види и завърши. Друг вариант е и двата процеса да извикат `sleep`. Това също ни осигурява тяхното съществуване достатъчно дълго, така че да могат да се видят един друг. При изпълнението на програмата получихме следния изход.

```
$ a.out
PID 24713: Child started, parent is 24712.
PID 24712: Started child PID 24713.
```

Друго, което процес-син наследява от бащата, са отворените файлове. Процесът-син получава копие на файловите дескриптори, които бащата е отворил преди да изпълни `fork`.

Пример

Програма 5.5 илюстрира наследяването на отворените файлове в процес-син.

```
/* ------------------------------------------------------------ */
/* Example of fork */

#include "ourhdr.h"

int glob = 5;
char buf[] = "write to standard output\n";

main(void)
{
  int local;
  pid_t pid;

  local = 77;

  if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
     err_sys_exit("write error");

  printf("before fork\n");  /* we do not flush stdout before fork */
  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0) {        /* in child */
       glob++;
       local++;
  } else                      /* in parent */
      sleep(3);

/* in parent and child*/
  printf("pid = %d, glob = %d, local = %d\n", getpid(), glob, local);
  exit(0);
}
/* ------------------------------------------------------------ */
```

При изпълнение на програмата получаваме следния изход.

```
$ a.out
write to standard output
before fork
pid = 1840, glob = 6, local = 78
pid = 1839, glob = 5, local = 77
$ a.out > out
$ cat out
write to standard output
before fork
pid = 1842, glob = 6, local = 78
before fork
pid = 1841, glob = 5, local = 77
```

Тук също използваме функцията `sleep` в процеса-баща, за да дадем възможност на сина да завърши. Този начин за синхронизация между двата процеса не е най-добрият, по-нататък ще разгледаме други механизми.

Забележете взаимодействието на `fork` с операциите по вход-изход. Тъй като при писане с `write` данните не се буферират в процеса, изходът от `write` преди `fork` се появява веднаж на стандартния изход. Изход чрез функции от стандартната В/И

библиотека обаче се буферира в процеса. При първото изпълнение на програмата стандартният изход е свързан с терминала и изходът от `printf` се изхвърля от буфера при символа за нов ред (line buffered). При второто изпълнение на програмата стандартният изход е пренасочен към файл `out` и изходът от `printf` се появява два пъти. Това е така защото се прави пълно буфериране на изхода от `printf` (fully buffered). Процесът-син, който е копие на бащата, получава и копие на буфера с данните и второто извикване на `printf` в сина добавя новите данните към получените от бащата. Когато двата процеса завършат съдържанието на буферите им се изхвърля.

Другото, което се вижда от този пример е, че когато пренасочим стандартния изход за процеса-баща, то се пренасочва и за процеса-син. Това е така защото, както казахме, синът наследява от бащата файловите дескриптори.

И така процес-син наследява от бащата следните атрибути и елементи от контекста си:

- идентификатор на група процеси
- идентификатор на сесия
- реален потребителски идентификатор
- ефективен потребителски идентификатор
- реален идентификатор на потребителска група
- ефективен идентификатор на потребителска група
- файлови дескриптори на отворените файлове
- текущ каталог
- управляващ терминал
- маска, използвана при създаване на файлове
- реакция на процеса при получаване на различни сигнали
- променливи от обкръжението

Това, по което се отличават процесите син и баща, е:

- значението, върнато от функцията `fork` в двата процеса е различно
- идентификатор на процеса
- идентификатор на процеса-баща
- времената за изпълнение на процеса-син в системна и потребителска фаза са 0

Има два основни начина за използване на `fork`.

1. Процес иска да създаде свое копие, което да изпълнява една операция, докато другото копие изпълнява друга. Този модел се използва при процеси сървери.
2. Процес иска да извика за изпълнение друга програма, тогава първо създава свое копие, след което в едното копие (обикновено в процеса-син) се извиква другата програма. Този начин се използва от командните интерпретатори при изпълнение на външни команди.

5.3. Завършване на процес - `exit` и `_exit`

Има няколко начина за завършване на процес:

1. Нормално завършване:
   - при извикване на `exit` или `_exit`
   - при изпълнение на `return` от функцията `main`, което е еквивалентно на извикването на `exit`
2. Аварийно завършване:
   - при получаване на сигнал, за който реакцията на процеса е завършване на процеса
   - при извикване на функцията `abort` (В този случай на процеса се изпраща сигнала `SIGABRT`.)

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

Обикновено `_exit` се реализира като системен примитив и предизвиква незабавен вход в ядрото, а `exit` е библиотечна функция. Тя осигурява „чисто" завършване на процеса. Грижи се да запише във файл буферите, използвани от функциите на стандартната В/И библиотека (напр., `printf`, `fwrite`), т.е. изпълнява `fclose` за всички незатворени потоци. Освен това извиква всички функции, регистрирани преди това с функцията `atexit`, наричани обработчици при завършване или exit handlers. След това извиква примитива `_exit`. Функциите `exit` и `atexit` са включени в ANSI C стандарта.

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Връща 0 при успех, -1 при грешка.

Аргументът *`function`* е адрес на функция, която ще бъде изпълнена при `exit`. Ако са регистрирани няколко функции, то те се изпълняват в ред обратен на реда на регистрацията им. Това се илюстрира в следващия пример.

Пример

Програма 5.6 демонстрира използването на функцията `atexit` и изпълнението на обработчици при завършване.

```
/* ------------------------------------------------------------ */
/* Example of exit handlers */

#include "ourhdr.h"

void my_exit1(void), my_exit2(void);

main(void)
{
  if (atexit(my_exit2) != 0)
     err_sys_exit("atexit my_exit2 error");
  if (atexit(my_exit1) != 0)
     err_sys_exit("atexit my_exit1 error");
  if (atexit(my_exit1) != 0)
     err_sys_exit("atexit my_exit1 error");
  printf("main is done\n");
  exit(0); /* or return(0); */
}

void my_exit1(void)
{
  printf("first exit handler\n");
}

void my_exit2(void)
{
  printf("second exit handler\n");
}
/* ------------------------------------------------------------ */
```

При изпълнението на програмата получаваме следния изход.

```
$ a.out
main is done
first exit handler
first exit handler
second exit handler
```

Независимо от начина, по който процесът завършва, накрая управлението се предава на ядрото. Там се освобождават почти всички елементи от контекста на процеса и от него остава само записа в таблицата на процесите. Функциите `exit` и `_exit` не връщат нищо, защото няма връщане от тях, винаги завършват успешно и след тях процесът почти не съществува, т.е. той става зомби. Значението на аргумента е кода на завършване на процеса, който се съхранява в таблицата на процесите. При аварийно завършване на процес, ядрото генерира код на завършване, който съобщава причината за аварийното завършване. Кодът на завършване е предназначен за процеса-баща и му се предава, когато той се заинтересува като извика примитив `wait`.

5. 4. Системни примитиви `wait`

Процес-баща може да разбере как е завършил негов процес-син, чрез примитив `wait`. Ако синът още не е завършил, то процесът-баща бива блокиран, докато синът не изпълни `exit`, т.е. изчаква неговото завършване.

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

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

Връща pid на процеса при успех, -1 при грешка.

Функциите на системния примитив връщат pid на завършилия син, а чрез аргумента *`status`* информация за начина, по който е завършил, т.е. кода на завършване. Различията между двете функции са следните:

1. Функцията `wait` чака първия син, който завърши. Функцията `waitpid` има аргумент *`pid`*, чрез който може да се чака завършването на определен син. Интерпретацията на аргумента *`pid`* е следната:
   - *`pid`* `== -1` Чака първия син, който завърши.
   - *`pid`* `> 0` Чака син с идентификатор *`pid`*.
   - *`pid`* `== 0` Чака първия син от същата група процеси.
   - *`pid`* `< -1` Чака първия син от група процеси с идентификатор |*`pid`*|.
2. Функцията `wait` винаги блокира процеса, ако синът не е завършил. Чрез аргумента *`options`* на `waitpid` може да се предотврати блокирането на процеса. При значение `WNOHANG`, процесът-баща не се блокира ако синът не е завършил, а функцията връща 0.

От значението, върнато в аргумента *`status`* на `wait` или `waitpid`, процес-баща може да разбере:

- Как е завършил сина - нормално или аварийно?
- Какъв е кода на завършване, предаден в `exit`, или номера на сигнала, който е предизвикал аварийното му завършване?
- Създаден ли е файл с име `core` (някои сигнали предизвикват създаването му)?

В заглавния файл `<sys/wait.h>` са определени няколко макроса, чрез които бащата може да определи как е завършил сина му. Тези макроси са показани в Ф*иг.5.1*.

Макрос	**Описание**
`WIFEXITED(status)`	Връща истина ако процесът-син (върнал `status`) е завършил нормално. В този случай кода на завършване, предаден от сина в `exit`, можем да извлечем чрез макроса `WEXITSTATUS(status).`

`WIFSIGNALED(status)`	Връща истина ако процесът-син е завършил аварийно - от сигнал.
	В този случай номера на сигнала можем да получим чрез макроса
	`WTERMSIG(status).`
	Дали е създаден файл `core` можем да определим чрез
	`WCOREDUMP(status).`

`WIFSTOPPED(status)`	Връща истина ако `status` е върнат от процес, който е в състояние stopped. Номера на сигнала можем да получим чрез макроса `WSTOPSIG(status).`

Фиг. 5.1. Макроси за проверка на *`status`*, върнат от `wait` или `waitpid`

Пример

Функцията `pr_estatus` използва тези макроси и извежда подробно описание за начина на завършване на процеса, който е върнал `status`. Тъй като макросът `WCOREDUMP` не е включен в стандарта POSIX и не се реализира във всички Unix системи, го използваме ако е определен. Програма 5.7 извиква функцията за различни значения на кода на завършване.

```
/* ------------------------------------------------------------ */
/* Example of exit status */

#include <sys/wait.h>
#include <signal.h>
#include "ourhdr.h"

/* Print the description of status */
void pr_estatus(int status)
{
  if (WIFEXITED(status))
     printf("normal termination, exit status = %d\n",
            WEXITSTATUS(status));
  else if(WIFSIGNALED(status))
     printf("abnormal termination, signal number = %d%s\n",
            WTERMSIG(status)
#ifdef WCOREDUMP
           , WCOREDUMP(status) ? " (core file)" : ""
#endif
            );
  else if(WIFSTOPPED(status))
     printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}
```

```
main(void)
{
  int status;
  pid_t pid;

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)
       exit(2);
  if (wait(&status) != pid)
     err_sys_exit("wait error");
  pr_estatus(status);

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)
       abort();        /* generates SIGABRT signal */
  if (wait(&status) != pid)
     err_sys_exit("wait error");
  pr_estatus(status);

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)
     pause();
  kill(pid, SIGTERM);   /* send SIGTERM signal to child */
  if (wait(&status) != pid)
     err_sys_exit("wait error");
  pr_estatus(status);

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)
       status /= 0;     /* generates SIGFPE signal */
  if (wait(&status) != pid)
     err_sys_exit("wait error");
  pr_estatus(status);

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0)  {
       alarm(3);       /* generates SIGALRM signal after 3 seconds */
       pause();
  }
  if (wait(&status) != pid)
     err_sys_exit("wait error");
  pr_estatus(status);

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

При изпълнението на програмата получаваме следния изход.

```
$ a.out
normal termination, exit status = 2
abnormal termination, signal number = 6 (core file)
abnormal termination, signal number = 15
abnormal termination, signal number = 8 (core file)
abnormal termination, signal number = 14
```

Процес-баща е отговорен за своите процеси-синове. Когато процес завърши се очаква неговият баща да се осведоми за завършването му с `wait`. При изпълнение на `wait` от процес-баща се изчиства сина-зомби от системата, т.е. освобождава се записа му от таблицата на процесите. Това означава, че всеки завършил процес остава в системата в състояние зомби докато баща му не изпълни `wait`. Но не бива да се задължава процес-баща да изпълнява `wait`, например той може да завърши веднага след като е създал син. Затова когато процес завършва неговите синове се осиновяват от процеса `init`, който се грижи за изчистване на системата от зомбита-сираци (така в системата няма да останат вечни зомбита).

Пример

Програма 5.8 показва как се създава зомби. Процесът-баща извиква функцията `sleep`, за да даде възможност на сина да завърши, след което изпълнява командата `ps`, която показва зомбито.

```
/* ------------------------------------------------------------ */
/* Creates a Zombie */

#include "ourhdr.h"

main(void)
{
  pid_t pid;

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");

  if (pid == 0) {   /* child process */
    printf("PID %d: Child started, parent is %d\n", getpid(), getppid());
    exit(0);
  } else {            /* parent process */
     printf("PID %d: Started child PID %d\n", getpid(), pid);
     sleep(5);        /* Wait 5 seconds */

/* By this time, our child process should have terminated */
     system("ps j"); /* List the zombie */
     exit(0);
  }
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида.

```
$ a.out
PID 15874: Child started, parent is 15873
PID 15873: Started child PID 15874
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     15873 Ss     501   0:00 -bash
12467 15873 15873 12467 tty1     15873 S+     501   0:00 a.out
15873 15874 15873 12467 tty1     15873 Z+     501   0:00 [a.out] <defunct>
15873 15877 15873 12467 tty1     15873 R+     501   0:00 ps j
```

Ако в този пример заменим `sleep` с `wait`, то командата `ps` не показва зомби.

Пример

Ако искаме програма, в която се създава процес-син, но бащата не чака завършването му и синът да не остава зомби, то може да използваме метода показан в следващата Програма 5.9. Процесът-внук (син от втория `fork`) извиква функцията `sleep`, за да даде възможност на първия син да завърши.

```
/* ------------------------------------------------------------ */
/* Avoid zombie process */

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

main(void)
{
  pid_t pid;

  printf("parent: pid=%d\n", getpid());
  if ((pid = fork()) < 0)
     err_sys_exit("first fork error");

  else if (pid == 0) {          /* first child */
       printf("first child: pid=%d, ppid=%d\n", getpid(), getppid());
       if ((pid = fork()) < 0)
          err_sys_exit("second fork error");
       else if (pid > 0)
            exit(0);    /* parent from second fork = first child exits */

       sleep(2);        /* child from second fork = grandchild */
       printf("grandchild: pid=%d, ppid=%d\n", getpid(), getppid());
       exit(0);
  }

  if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
     err_sys_exit("waitpid error");

/* original process continue executing, knowing it's not parent */
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида.

```
$ a.out
parent: pid=1900
first child: pid=1901, ppid=1900
$ grandchild: pid=1902, ppid=1
```

5.5. Функции на системния примитив `exec`

Когато с `fork` се създава нов процес, той е копие на баща си, т.е. продължава да изпълнява същата програма. Чрез примитива `exec` всеки процес може да извика за изпълнение друга програма по всяко време от своя живот, както веднага след създаването си така и по-късно, дори няколко пъти в живота си. За този системен примитив има няколко функции в стандартната библиотека, които се различават по начина на предаване на аргументите и използване на променливите от обкръжението на процеса.

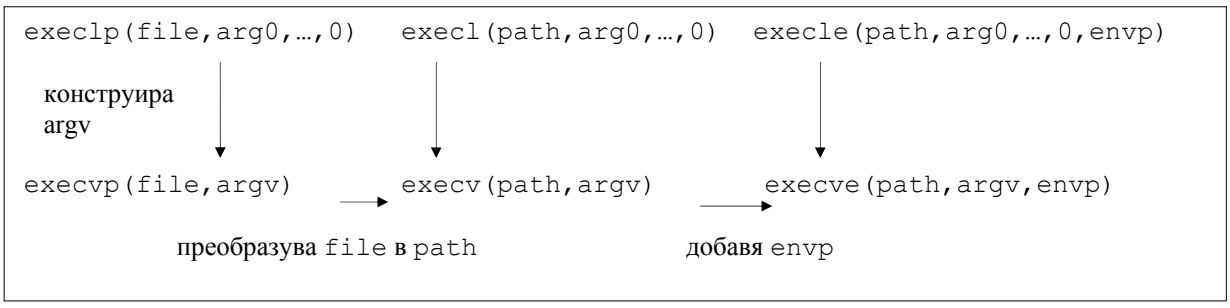
```
#include <unistd.h>

int execl(const char *name, const char *arg0
            [, const char *arg1]..., 0);

int execlp(const char *name, const char *arg0
            [, const char *arg1]..., 0);

int execv(const char *name, char *const argv[]);

int execvp(const char *name, char *const argv[]);

int execle(const char *name, const char *arg0
            [, const char *arg1]..., 0, char *const envp[]);

int execve(const char *name, char *const argv[], char *const envp[]);
```

Връщат -1 при грешка, нищо при успех.

Връзката между функциите на примитива `exec` е показана на *Фиг.5.2*. В трите функции от първия ред, всеки аргумент на `main` е зададен като отделен аргумент на функцията `exec`. В трите функции от долния ред има един аргумент `argv`, който е масив от указатели към аргументите за `main`.

В двете функции в ляво `file` може да е собствено име на файл, което се преобразува в пълно име чрез значението на променливата `PATH`. В останалите функции `path` трябва да е пълно име на файл.

Двете функции в дясно имат аргумент `envp`, който е масив от указатели към символни низове, съдържащи променливите от обкръжението на процеса. В четирите функции в ляво няма аргумент за променливите от обкръжението и се използва значението на глобалната променлива `environ`.



Фиг. 5.2. Връзка между функциите на примитива `exec`

При успех, когато процесът се върне от `exec` в потребителска фаза, той изпълнява кода на новата програма, започвайки от функцията `main`, но това си остава същия процес. Не е променен идентификатора му, позицията му в йерархията на процесите дори голяма част от елементи на контекста му. При грешка по време на `exec`

става връщане в стария образ, така че функцията връща -1 при грешка, а при успех не връща нищо защото няма връщане в стария образ.

Пример

Програма 5.10 демонстрира функции на примитива `exec`. Програмата `echoall`, която се извиква за изпълнение в процесите-синове, е тази от Пример 5.2. Извиква се два пъти, първя път с `execle` и втория с `execlp`.

```
/* ------------------------------------------------------------ */
/* Example of exec */

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

char *env_list[] = { "USER=test", "PATH=/tmp", "X=123", (char *)0 };

main(void)
{
  pid_t pid;
  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0) {        /* specify environment */
          execle("/home/moni/bin/echoall",
                 "echoall", "myarg1", "MyARG2", (char *)0, env_list);
          err_sys_exit("execle error");
  }
  if (waitpid(pid, NULL, 0) < 0)
      err_sys_exit("wait error");

  if ((pid = fork()) < 0)
      err_sys_exit("fork error");
  else if (pid == 0) {        /* inherit environment */
          execlp("echoall",
                 "echoall", "argument 1", (char *)0);
          err_sys_exit("execlp error");
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида.

```
$ a.out
arg[0]: echoall
arg[1]: myarg1
arg[2]: MyARG2
USER=test
PATH=/tmp
X=123

arg[0]: echoall
arg[1]: argument 1
HOME=/home/moni
SHELL=/bin/bash
PS1=[\u@\h \W]\$
USER=moni                <и още много други променливи, които не са показани>
```

И така след успешно изпълнение на примитива `exec` основно се променя образа на процеса, но освен това се променят следните атрибути и елементи от контекста на процеса:

- променливите от обкръжението, ако се използва функция `execle` или `execve`, в която е указан аргумент `envp`;

- ефективния потребителски идентификатор - ако е вдигнат set-UID бита за файла с име *name*;
- ефективния идентификатор на потребителска група - ако е вдигнат set-GID бита за файла с име *name*;
- файлови дескриптори на отворените файлове - затварят се тези с вдигнат флаг `FD_CLOEXEC`;
- реакция на процеса при получаване на сигнали - за сигналите, за които реакцията е потребителска функция, се връща реакцията по премълчаване.

5.6. Потребителски идентификатори на процес

С всеки процес са свързани два потребителски идентификатора: реален - ruid и ефективен - euid. Реалният е идентификатора на потребителя, който е създал процеса, а по ефективния се определят правата на процеса при работа с файлове, при изпращане на сигнали и др. Обикновено ефективният е еднакъв с реалния, освен ако не е променен. В същност се пази още един ефективен идентификатор, наричан съхранен uid - suid, който се използва, за да може да се възстанови временно изменен euid.

Всеки процес може да научи потребителските си идентификатори. Системният примитив `getuid` връща реалния потребителски идентификатор на процеса, а `geteuid` - ефективния потребителски идентификатор. Аналогично, примитивите `getgid` и `getegid` връщат реалния и ефективния идентификатори на потребителска група на процеса. Тези системни примитиви винаги завършват успешно.

```
#include <unistd.h>
uid_t getuid(void);
                                                    Връща ruid на процеса.
uid_t geteuid(void);
                                                    Връща euid на процеса.
gid_t getgid(void);
                                                    Връща rgid на процеса.
gid_t getegid(void);
                                                    Връща egid на процеса.
```

Процес-син наследява потребителските идентификатори от процеса-баща. Промяна на ефективния потребителски идентификатор може да стане по два начина.

Когато се изпълнява системен примитив `exec` и програмата е set-UID се променя ефективния потребителски идентификатор. Една програма се нарича set-UID, когато в кода на защита на файла с изпълнимия код битът "изменение на UID при изпълнение" е 1 (т.е. 04000). Тогава при `exec` се сменя euid и suid на процеса със собственика на файла с изпълнимия код. Това означава, че по време на изпълнение на новата програма процесът ще има правата на собственика на файла с програмата. Този начин се използва от някои команди за контролирано и временно повишаване на правата на потребителите. Например, set-UID програма е командата `passwd`, чрез която всеки потребител може да смени паролата си, т.е. да пише във файла с паролите. Файлът на командата `passwd` е собственост на `root`, следователно процесът, в който се изпълнява тя има правата на администратора.

Пример

Програма 5.11 е пример за set-UID програма. Тя покава и наследяването на потребителските идентификатори при пораждане на процеси.

```
/* ------------------------------------------------------------ */
/* Example of set-UID program */

#include <stdio.h>

main(void)
{
  pid_t pid;
  int status;

  pid = fork();
  if (pid == 0){
     printf("Child:  pid=%d, ruid=%d, euid=%d\n",
             getpid(), getuid(), geteuid());
  }
  else {
     printf("Parent: pid=%d, ruid=%d, euid=%d\n",
             getpid(), getuid(), geteuid());
     wait(&status);
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Изпълнимият файл `a.out` е собственост на `moni` и е с вдигнат set-UID бит (с код на защита 04755). Изпълняваме програмата два пъти, първия път в сесия на потребителя `moni`, втория от сесия на друг потребител - `ivan`. Получихме следния изход.

```
$ a.out                          <от сесия на moni (501)>
Parent: pid=635, ruid=501, euid=501
Child:  pid=636, ruid=501, euid=501
$ a.out                          <от сесия на ivan (509)>
Parent: pid=637, ruid=509, euid=501
Child:  pid=638, ruid=509, euid=501
```

При първото изпълнение няма разлика между реалния и ефективния потребителски идентификатори. При второто изпълнение процесите са с променен ефективен потребителски идентификатор. Другото, което се вижда и при двете изпълнения, е че процес-син наследява от баща се потребителските идентификатори.

Другият начин за промяна на потребителските идентификатори е чрез системен примитив `setuid`.

```
#include <unistd.h>

int setuid(uid_t uid);
```

Връща 0 при успех, -1 при грешка.

Ако текущият euid на процеса е 0 (на `root`), то се променя euid, ruid и suid с аргумента *`uid`*. Ако текущият euid на процеса не е 0, то се променя euid с аргумента *`uid`*, само ако аргументът *`uid`* е равен на текущия ruid или на suid на процеса.

Пример

Програма 5.12 илюстрира примитива `setuid`.

```
/* ------------------------------------------------------------ */
/* Example of setuid() */

#include <fcntl.h>
```

```
main(void)
{
  int uid, euid, fdm, fdi;

  uid = getuid();
  euid = geteuid();
  printf("ruid=%d, euid=%d\n", uid, euid);

  fdm = open("file_moni", O_RDONLY);
  fdi = open("file_ivan", O_RDONLY);
  printf("fdm=%d, fdi=%d\n", fdm, fdi);

  setuid(uid);
  printf("after setuid(%d): ruid=%d, euid=%d\n",
         uid, getuid(), geteuid());

  fdm = open("file_moni", O_RDONLY);
  fdi = open("file_ivan", O_RDONLY);
  printf("fdm=%d, fdi=%d\n", fdm, fdi);

  setuid(euid);
  printf("after setuid(%d): ruid=%d, euid=%d\n",
         euid, getuid(), geteuid());
  exit(0);
}
/* ------------------------------------------------------------ */
```

Изпълнимият файл `a.out` е собственост на потребител `moni` и е с вдигнат set-UID бит. Предполагаме, че в текущия каталог има файл `file_moni`, който е собсвеност на `moni` и файл `file_ivan`, собсвеност на `ivan`. Изпълняваме програмата два пъти, първия път в сесия на потребителя `moni`, втория от сесия на потребителя `ivan`. Получихме следния изход.

```
$ ls -ln file*
-rw-------  1 509 500 0 Sep  7 17:04 file_ivan
-rw-------  1 501 500 0 Sep  7 17:03 file_moni
$ a.out                 <от сесия на moni (501)>
ruid=501, euid=501
fdm=3, fdi=-1
after setuid(501): ruid=501, euid=501
fdm=4, fdi=-1
after setuid(501): ruid=501, euid=501
$ a.out                 <от сесия на ivan (509)>
ruid=509, euid=501
fdm=3, fdi=-1
after setuid(509): ruid=509, euid=509
fdm=-1, fdi=4
after setuid(501): ruid=509, euid=501
```

5.7. Групи процеси и сесия

Всеки процес принадлежи на група от процеси, която включва един или повече процеса. Всяка група има лидер на групата (group leader), който е процесът, създал групата. Групата съществува докато съществува поне един от процесите в нея, независимо дали лидерът е завършил или не. Последният процес от групата може или да завърши или да премине към друга група. Групата се идентифицира чрез идентификатор на група процеси (process group ID или PGID), който в същност е pid на процеса-лидер на групата. Следователно, всеки процес има и идентификатор на група процеси, ще го наричаме групов идентификатор за по-кратко. Процес-син наследява

груповия идентификатор от процеса-баща, а при `exec` груповият идентификатор не се променя.

Следват системните примитиви свързани с понятието група процеси.

```
#include <unistd.h>
pid_t getpgrp(void);
                                        Връща груповия идентификатор на процеса.
pid_t getpgid(pid_t pid);  /* SVr4 */
                                    Връща групов идентификатор при успех, -1 при грешка.
```

Системният примитив `getpgrp` връща груповия идентификатор на процеса, който го изпълнява, т.е. всеки процес може да научи към коя група принадлежи. Системният примитив `getpgid` връща груповия идентификатор на процес с идентификатор *pid*. Ако *pid* е 0, то се връща групата на текущия процес. Процесът, изпълняващ системния примитив трябва да принадлежи на сесията, към която принадлежи и процеса *pid*. При грешка `getpgid` връща -1, а `getpgrp` винаги завършва успешно.

Процес може да смени групата, към която принадлежи, като създаде своя група или се присъедини към друга съществуваща група, чрез системните примитиви:

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
int setpgrp(void);                      /* BSD4.2 */
                                          Връщат 0 при успех, -1 при грешка.
```

Системният примитив `setpgrp` създава нова група и процесът, който го изпълнява става неин лидер, т.е. групов идентификатор на процеса става неговия pid. При `setpgid` процес с идентификатор *pid* преминава към група *pgid*. Ако *pid* е 0, то се използва идентификатора на текущия процес, а ако *pgid* е 0, то се използва идентификатора *pid*. Чрез този примитив процес може да смени групата за себе си или процес-баща да смени групата за свой процес-син. Когато процес се мести в друга съществуваща група, двете групи - старата и новата, трябва да са в една сесия. Примитивът `setpgid(0,0)` има същото действие както `setpgrp()`. При успех и двете функции връщат 0.

Пример

Програма 5.13 илюстрира наследяването на групата и създаването на нова група.

```
/* ------------------------------------------------------------ */
/* Example of setgrp() */

#include "ourhdr.h"

main(void)
{
  pid_t pid;
  int status;

  printf("Parent: pid=%d, grp=%d\n",
             getpid(), getpgrp());

  if ((pid = fork()) < 0)
```

```
     err_sys_exit("fork error");
  else if (pid == 0){
     printf("Child:  pid=%d, grp=%d\n",
             getpid(), getpgrp());

/* child become group leader */
     if (setpgrp() == -1)  /* or setpgid(0,0) */
        err_sys_exit("setpgrp error");
     printf("Child after setpgrp: pid=%d, grp=%d\n",
             getpid(), getpgrp());
     system("ps j");
     exit(0);
  }

  wait(&status);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата получихме следния изход.

```
$ a.out
Parent: pid=16235, grp=16235
Child:  pid=16236, grp=16235
Child after setpgrp: pid=16236, grp=16236
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     16235 Ss     501   0:00 -bash
12467 16235 16235 12467 tty1     16235 S+     501   0:00 a.out
16235 16236 16236 12467 tty1     16235 S      501   0:00 a.out
16236 16237 16236 12467 tty1     16235 R      501   0:00 ps j
```

След `fork` процесът-син е в групата на процеса-баща. След като изпълни `setpgrp`, той става лидер на нова група. Това се вижда и от изхода на командата `ps`.

Понятието сесия е въведено в Unix системите с цел логическо обединение на процесите, създадени в резултат на login и последващата работа на един потребител. Сесията включва една или повече групи процеси. Всяка сесия има лидер на сесия, който е процесът, създал сесията. Аналогично на групите, сесията се идентифицира чрез идентификатор на сесия (session ID или SID), който в същност е pid на процеса-лидер на сесията. Следователно, всеки процес притежава и идентификатор на сесията, който наследява от процеса-баща, а при `exec` идентификатора на сесия не се променя.

Следват системните примитиви свързани с понятието сесия.

```
#include <unistd.h>

pid_t getsid(pid_t pid);

pid_t setsid(void);
```

Връщат идентификатор на сесия при успех, -1 при грешка.

Системният примитив `getsid` връща идентификатора на сесия за процес с идентификатор *`pid`*. Ако *`pid`* е 0, то се използва идентификатора на текущия процес. Процес с правата на `root` може да изпълни примитива за всеки друг процес, но за процес с обикновени права *`pid`* трябва да е идентификатор на процес от същата сесия.

Нова сесия се създава с `setsid` ако процесът, изпълняващ примитива, преди това не е лидер на група. При успех процесът, изпълняващ примитива става лидер на новата сесия, лидер на първата група в тази сесия и няма управляващ терминал. При успех примитивът връща идентификатора на новата сесия, а при грешка връща -1.

Понятията група и сесия са тясно свързани с понятието терминал. Също така се използват и от механизма на сигналите. Всяка сесия може да има свързан с нея терминал, който се нарича управляващ терминал на процеса (controlling terminal). Това позволява на ядрото да контролира стандартния вход и изход на процесите, а също и да им изпраща сигнали за събития, свързани с терминала.

Всеки процес има поле за управляващ терминал в потребителската област. Какво съдържа това поле? Всеки терминал има свързана с него tty структура. Полето за управляващ терминал на процеса е указател към tty структурата на управляващия му терминал. Ако това поле има значение `NULL`, то процесът няма управляващ терминал.

Как и кога се свързва управляващ терминал с процес? Обикновено ние не трябва да се тревожим за управляващия терминал, тъй като го получаваме при login. Всеки процес наследява управляващия терминал от процеса-баща при `fork`, а при `exec` управляващият терминал не се променя. Процесът `getty` (`mingetty` в Linux) има управлявляващ терминал, който се наследява от login-shell процеса, който става лидер на сесия. В някои версии на Linux (като Linux Kernel 2.0, която сме използвали при теста на някои от примерите) лидер на сесия е процесът `login`, а негов син е процесът `bash`, който принадлежи на същата сесия. След това всички процеси, пораждани от login-shell процеса при изпълнение на команди наследяват от него идентификатора на сесия и управляващия терминал, т.е. принадлежат на една сесия и са свързани с един управляващ терминал.

Но как процесът `getty` е получил управляващ терминал? Процес лидер на сесия установява връзка с управляващ терминал. Начинът, по който се установява връзка с управляващ терминал е системно зависим. Например, в UNIX System V и Linux ядрото свързва управляващ терминал с процес, когато той изпълнява примитива `open` на специален файл за терминал (напр., `fd=open("/dev/tty1", …);`) и ако са изпълнени условията:

- Терминалът в момента не е управляващ терминал на сесия.
- Процесът е лидер на сесия.

Следващият въпрос е как се прекъсва връзката на процес с управляващия му терминал? Това пак е различно в различните Unix и Linux системи. В Linux това става с примитива `setsid`, когато се изпълнява от процес, който не е лидер на група. Тогава той става лидер на нова група, на нова сесия и губи управляващия си терминал.

Пример

Програма 5.14 илюстрира наследяването на сесия и създаването на нова сесия.

```
/* ------------------------------------------------------------ */
/* Example of setsid()*/

#include "ourhdr.h"

main(void)
{
  pid_t pid;
  int status;

  printf("Parent: pid=%d, grp=%d, sid=%d\n",
          getpid(), getpgrp(), getsid(0));

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid == 0){
     printf("Child:  pid=%d, grp=%d, sid=%d\n",
             getpid(), getpgrp(), getsid(0));
```

```
 /* child become session and group leader without tty */
     if (setsid() == -1)
        err_sys_exit("setsid error");
     printf("Child after setsid: pid=%d, grp=%d, sid=%d\n",
             getpid(), getpgrp(), getsid(0));

     system("ps xj");
     exit(0);
   }

   wait(&status);
   exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата в Linux Kernel 2.0 получихме следния изход.

```
$ a.out
Parent: pid=931, grp=931, sid=140
Child:  pid=932, grp=931, sid=140
Child after setsid: pid=932, grp=932, sid=932
 PPID   PID  PGID   SID TTY TPGID  STAT   UID   TIME COMMAND
    1   140   140   140   1   931  S     501   0:00 /bin/login -- moni
  140   153   153   140   1   931  S     501   0:00 -bash
  153   931   931   140   1   931  S     501   0:00 a.out
  931   932   932   932  ?     -1  S     501   0:00 a.out
  932   933   932   932  ?     -1  R     501   0:00 ps xj
```

След `fork` процесът-син е в сесията на процеса-баща. След като изпълни `setsid`, той става лидер на нова сесия и лидер на първата група в сесията. Освен това, от изхода на командата `ps` се вижда, че процесът няма управляващ терминал.

В Linux Kernel 2.6 получихме подобен изход.

```
Parent: pid=16261, grp=16261, sid=12467
Child:  pid=16262, grp=16261, sid=12467
Child after setsid: pid=16262, grp=16262, sid=16262
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     16261 Ss     501   0:00 -bash
12467 16261 16261 12467 tty1     16261 S+     501   0:00 a.out
16261 16262 16262 16262 ?           -1 Ss     501   0:00 a.out
16262 16263 16262 16262 ?           -1 R      501   0:00 ps xj
```

Когато сесията има управляващ терминал, групите в нея са: една привилегирована (foreground group) и всички други - фонови групи (background group). В tty структура на управляващия терминал има поле, което съдържа идентификатор на група процеси (terminal group ID или TPGID) - привилегированата в момента група. Това поле определя групата процеси, на които се изпращат сигнали, свързани с терминала: `SIGINT`, `SIGQUIT`, `SIGHUP`, `SIGTSTP`. Така, когато въведем <Ctrl>+<C> на терминала, ядрото изпраща сигнал `SIGINT` на всички процеси от привилегированата група. Освен това, входът от терминала също се изпраща към процесите от привилегированата група.

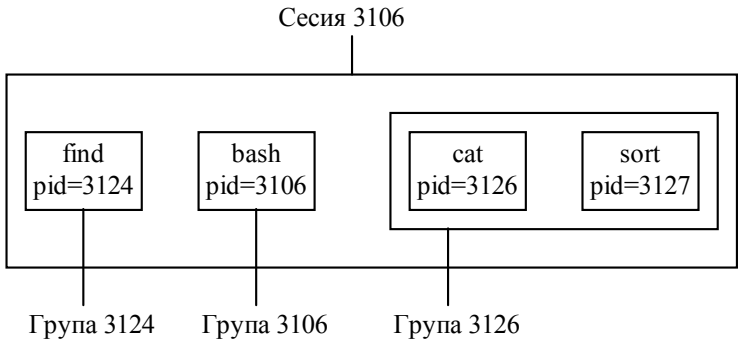
При `exit` на процес-лидер на сесия с управляващ терминал, се прекъсва връзката между сесията и терминала, т.е. сесията вече няма управляващ терминал. Също така се изпраща сигнал `SIGHUP` на процесите от привилегированата група.

5.8. Управление на заданията

Управление на заданията (Job control) е възможност добавена в BSD към 1980г. и приета в POSIX. Някои командни интерпретатори, като Korn shell, C shell и Bash реализират понятието задание (наричат ги job control shell) като използват групи процеси. Други командни интерпретатори, като B shell не поддържат задания. Заданието се реализира като един или повече процеса, които са в една група. Всички процеси пораждани от login-shell процеса са организирани като задания, т.е. групирани в групи, като една от тях е привилегирована, а всички останали са фонови групи. Когато поддържащ заданията shell стартира ново задание в привилегирован режим, сменя полето TPGID в tty структурата на управляващия терминал да съответства на това задание. (Неподдържащ заданията shell никога не сменя TPGID, то винаги съдържа pid на login- shell процеса.)

Да разгледаме изпълнението на следващите команди от поддържащ заданията Bash (Ф*иг.5.3а*).

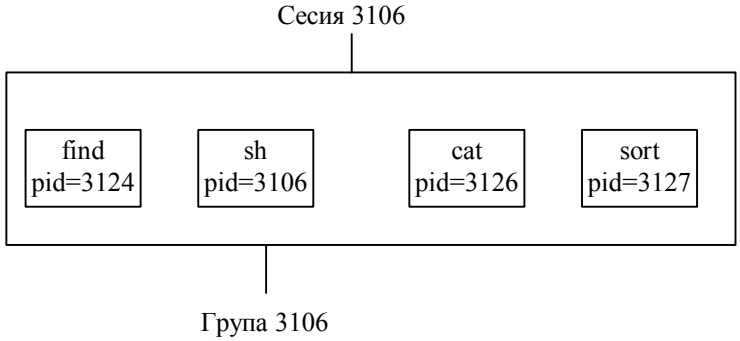
```
$ find /home -name core -print  > outfind 2>&1 &
$ cat | sort
```



Фиг. 5.3а. Сесия и групи процеси

Всичките четири процеса `bash`, `find`, `cat` и `sort` принадлежат на една сесия с лидер процеса `bash` (обикновено е така, но може да е и процеса `login`) и имат един и същи управляващ терминал. Процесите `cat` и `sort` принадлежат на една група с лидер `cat`, която в момента е привилегирована, процесът `find` е лидер на друга група и `bash` е лидер на третата група.

При неподдържащ заданията shell, всичките четири процеса `sh`, `find`, `cat` и `sort` принадлежат на една група с лидер процеса `sh` и на една сесия със същия лидер и имат един и същ управляващ терминал (*Фиг.5.3б*).



Фиг. 5.3б. Сесия и групи процеси в B shell

Пример

Програма 5.15 ще ни покаже с какъв shell работим.

```
/* ------------------------------------------------------------ */
/* Job control test */
main(void)
{
  printf("pid=%d, pgrp=%d\n", getpid(), getpgrp());
}
/* ------------------------------------------------------------ */
```

Ако изпълним тази програма в неподдържащ заданията B shell, ще получим резултати, подобни на следните.

$ **job_test**	pid=3128, pgrp=3106
$ **job_test & job_test**	pid=3129, pgrp=3106 pid=3130, pgrp=3106
$ **(job_test & job_test)**	pid=3132, pgrp=3106 pid=3133, pgrp=3106

Процесът login-shell е с pid 3106. В последния случай, липсващия pid 3131 е на процеса subshell. И в трите случая породените процеси са в една група с лидер процеса login-shell.

Ако изпълним тази програма в поддържащ заданията Bash, ще получим резултати, подобни на следните.

$ **job_test**	pid=3128, pgrp=3128
$ **job_test & job_test**	pid=3129, pgrp=3129 pid=3130, pgrp=3130
$ **(job_test & job_test)**	pid=3132, pgrp=3131 pid=3133, pgrp=3131

Всеки път програмата се изпълнява в собствена група. В последния случай, резултатите може да са различни, в зависимост от това дали процес subshell (pid 3131) поддържа задания. В случая двата процеса са в една група с лидер процеса subshell, т.е. процес subshell не поддържа задания.

Шеста глава

СИГНАЛИ

Сигналите информират процес за настъпване на асинхронни събития вън от процеса или на особени събития в самия процес. Сигналите могат да се разглеждат като най-примитивния механизъм за междупроцесни комуникации, но също така те много напомнят механизма на прекъсвания. Сигналите се появяват още в най-ранните версии на Unix, но в реализацията им има някои недостатъци. В следващите версии на BSD и UNIX System V са внесени изменения, но моделите в двете версии са несъвместими, затова всички версии на Unix и Linux поддържат и първоначалната семантиката на "ненадеждните" сигнали. По нататък ние ще разглеждаме именно тези сигнали.

6.1. Типове сигнали

Всеки сигнал има уникален номер и символно име, които определят събитието, за което информира сигнала. В ранните версии има около 15 типа сигнала, а в новите броят им е около 30. Типовете сигнали могат да се класифицират в зависимост от събитието, свързано със сигнала:

- Сигнали, свързани с управляващия терминал

Изпращат се на процеса (процесите) от привилегированата групата, свързан с терминала.

`SIGINT` Изпраща се когато потребителят натисне клавиша <Del> или <Ctrl>+<C>.

`SIGQUIT` Изпраща се когато потребителят натисне клавишите <Ctrl>+<\>.

`SIGHUP` Изпраща се при прекъсване на връзката с управляващия терминал.

- Сигнали свързани с апаратни особени ситуации

Сигналите в тази категория са свързани със събития, откривани от апаратурата и сигнализирани чрез прекъсване. Ядрото реагира на това като изпраща сигнал на процеса, който е текущ в момента. Някои от типовете и събитията са:

`SIGFPE` Изпраща се при деление на 0 или препълване при операции с плаваща точка.

`SIGILL` Изпраща се при опит за изпълнение на недопустима инструкция.

`SIGSEGV` Изпраща се при обръщение към недопустим адрес или към адрес, за който процесът няма права.

- Сигнали свързани с програмни ситуации

Сигналите в тази категория са свързани с най-различни събития, синхронни или асинхронни с процеса, на който са изпратени, които имат чисто програмен характер и не се сигнализират от апаратурата. Някои от типовете сигнали са:

`SIGCHLD` Изпраща се на процес-баща когато някой негов процес-син завърши.

`SIGALRM` Изпраща се когато изтече времето, заредено от процеса, чрез системния примитив `alarm`.

`SIGPIPE` Изпраща се при опит на процес да пише в програмен канал, който вече не е отворен за четене.

Един процес може да изпрати на друг процес сигнал чрез системния примитив `kill`. Някои от типовете сигнали, които могат да бъдат изпратени само чрез `kill` са:

`SIGKILL` Предизвиква безусловно завършване на процеса.

`SIGTERM` Предупреждение за завършване на процеса.

`SIGUSR1` Потребителски сигнал, използван от потребителските процеси като средство за междупроцесни комуникации.

`SIGUSR2` Още един потребителски сигнал.

6.2. Изпращане и обработка на сигнали

Сигнал може да бъде изпратен на процес или от ядрото или от друг процес чрез системния примитив `kill`. Ядрото помни изпратените, но още необработени от процеса сигнали в записа от таблицата на процесите. Полето е масив от битове, в който всеки бит отговаря на тип сигнал. При изпращане на сигнал съответния бит се вдига. С това работата по изпращане е завършена.

Обработката на изпратените сигнали се извършва в контекста на процеса, получил сигнала, когато процесът се връща от системна в потребителска фаза. Следователно, сигналите нямат ефект върху процес, работещ в системна фаза докато тя не завърши. Има три възможни начина да бъде обработен един сигнал, ще ги наричаме реакции на сигнал:

- Процесът завършва (това е реакцията по премълчаване за повечето типове сигнали).
- Сигналът се игнорира.
- Процесът изпълнява определена потребителска функция, след което продължава изпълнението си от мястото където е бил прекъснат от сигнала.

Реакцията за всички типове сигнали се помни в потребителската област на процеса, където полето е масив от адресите на обработчиците на сигналите, един за всеки тип сигнал.

Процес може да определи реакцията си при получаване на сигнал от определен тип, ако иска тя да е различна от тази по премълчаване, чрез системния примитив `signal`.

```
#include <signal.h>
void (*signal(int sig, void (*sighandler)(int)))(int);
                    Връща адреса на предишната реакция при успех, -1 при грешка.
```

Аргументът *`sig`* задава номера на сигнала, а *`sighandler`* определя каква да е реакцията при на получаване на сигнал. Значението на втория аргумент може да е едно от следните:

- `SIG_IGN` - игнориране на сигнал;
- `SIG_DFL` - реакция по премълчаване, която за повечето типове сигнали е завършване на процеса;
- име на потребителската функция.

Реакцията се запомня в съответното поле от потребителската област и с това работата на примитива приключва. При успех функцията връща адреса на предишната реакция, който може да бъде запомнен и по-късно възстановен, а при грешка връща -1.

Процес-син наследява реакциите на сигнали от процеса-баща. След `exec` за всички сигнали, за които реакцията е била променена с потребителска функция, се връща реакцията по премълчаване. Това е естествено поведение, тъй като при `exec` се сменя образа на процеса.

Когато по-късно пристигне сигнал от съответния тип, той ще бъде обработен според запомнената реакция. Ако това е била потребителска функция, то преди обработката се връща реакцията по премълчаване. Това е семантиката в UNIX System V, в BSD не се прави такова връщане. Следователно, ако процес иска да обработва повтарящи се сигнали от един тип чрез потребителска функция, трябва отново да изпълнява `signal` след всеки получен сигнал. Това решение изглежда вярно, защото в

повечето случаи работи правилно, но не е. Нов сигнал може да пристигне преди процесът да успее да изпълни `signal`, тъй като когато процес работи в потребителска фаза, ядрото може да направи превключване на контекста преди процесът да е стигнал до извикването на `signal`. Затова се препоръчва извикването на `signal` да е в началото на функцията, обработваща сигнала (въпреки, че това не решава проблема с повтарящите се сигнали).

Сега вече се виждат недостатъците в семантиката на ненадеждните сигнали.

- Може да се получи състезание при повтарящи се сигнали от един тип.
- Може да има загуба на сигнали, тъй като няма памет, в която да се помнят няколко изпратени сигнала от един тип.
- Процес не може да блокира и след това разблокира, получаването на сигнали за известно време, като ядрото да помни изпратените през това време сигнали (подобно на апаратните прекъсвания).
- Процес не може да провери реакцията си за определен тип сигнал без да я променя.

Пример

Програмата 6.1 илюстрира механизма на сигналите. Два типа сигнала `SIGUSR1` и `SIGUSR2` се обработват с една и съща потребителска функция, която извежда съобщение и връща управлението.

```
/* ------------------------------------------------------------ */
/* Catch SIGUSR1 and SIGUSR2 */

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

void sig_usr(int);

main(void)
{
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
     err_sys_exit("can't catch SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
     err_sys_exit("can't catch SIGUSR2");

  for ( ; ; )
      pause();
}

void sig_usr(int sig)
{
  if (sig == SIGUSR1)
     printf("received SIGUSR1\n");
  else if (sig == SIGUSR2)
     printf("received SIGUSR2\n");
  else
     err_exit("received signal %d\n", sig);
  return;
}
/* ------------------------------------------------------------ */
```

Изпълняваме програмата във фонов режим и извикваме командата `kill`, с която изпращаме сигнали на процеса. Сигналите `SIGUSR1` и `SIGUSR2` се прихващат от процеса и той продължава работата си, но сигналът `SIGTERM` убива процеса, тъй като за него реакцията е по-премълчаване.

```
$ a.out &
[1] 445
$ kill -USR1 445
received SIGUSR1
$ kill -USR2 445
received SIGUSR2
$ kill 445
[1]+  Terminated              a.out
```

Изпращане на сигнал от един процес към друг(и) се извършва с примитива `kill`.

```
#include <signal.h>
int kill(pid_t pid, int sig);
                                        Връща 0 при успех, -1 при грешка.
```

Аргументът *sig* задава номера на типа сигнал, който се изпраща. Аргументът *pid* определя процесите, на които се изпраща сигнала. Възможните значения на *pid* са:

- *pid* > 0 Сигнал се изпраща на процеса с идентификатор *pid*.
- *pid* = 0 Сигнал се изпраща на всички процеси от групата на процеса, изпращащ сигнала.
- *pid* < -1 Сигнал се изпраща на всички процеси от групата с идентификатор |*pid*|.
- *pid* = -1 Сигнал се изпраща на всички процеси в таблицата на процесите, започвайки от процеса с най-голям идентификатор, без процеса `init` (с pid 1) и системните процеси.

Във всичките случаи е необходимо процесът, изпращащ сигнала, да има права, т.е. да е изпълнено едно от условията:

- Ефективният потребителски идентификатор (euid) на процеса, изпращащ сигнала, да е 0.
- ruid или euid на процеса, изпращащ сигнала, да е еднакъв с ruid или suid на процеса, на когото се изпраща сигнала.

В противен случай сигнал не се изпраща и примитивът връща -1. При успех връща 0. Ако аргументът *sig* е 0, сигнал не се изпраща, но се прави проверка за грешка.

Системният примитив `pause` блокира процеса, който го изпълнява до получаването на първия сигнал, за който реакцията не е игнориране.

```
#include <unistd.h>
int pause(void);
                                                    Връща -1.
```

Ако реакцията за първия пристигнал сигнал е `SIG_DFL`, то процесът завършва, т.е. връщане от `pause` няма. Ако за сигнала е предвидена потребителска функция и от тази функция има връщане, то след изпълнение на функцията процесът продължава от оператора след `pause`. В този случай има връщане от `pause` и функцията връща -1.

Системният примитив `alarm` планира изпращането на сигнал `SIGALRM` на процеса, изпълняващ примитива.

```
#include <unistd.h>
unsigned int alarm(unsigned int sec);
                                        Връща 0 или число по-голямо от 0.
```

Аргументът *sec* задава брой секунди, т.е. при изпълнение на `alarm` в таймера на процеса се зарежда значението на аргумента *sec*. Когато изтече този интервал от време ядрото ще изпрати сигнал `SIGALRM` на процеса. Ако преди това е било планирано друго изпращане на сигнал `SIGALRM`, което не се е състояло, то се анулира и примитивът връща броя секунди, оставащи до него. В противен случай функцията връща 0. Чрез изпъление на примитива с аргумент *sec* равен на 0 може да се анулира зареден преди това таймер без да се планира нов сигнал.

Пример

Програмата 6.2 реализира функция `mysleep`, наш и несъвършен вариант на библиотечната функция `sleep`. Два типа сигнали `SIGINT` и `SIGALRM` се обработват с различни потребителски функции.

```
/* ------------------------------------------------------------ */
/* Implementation of sleep function */

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

void sig_int(int);
void sig_alrm(int);
unsigned int mysleep(unsigned int);

main(void)
{
  unsigned int ret_sl;

  if (signal(SIGINT, sig_int) == SIG_ERR)
     err_sys_exit("can't catch SIGINT");

  ret_sl = mysleep(4);

  printf("mysleep returned %u\n", ret_sl);
  exit(0);
}

void sig_int(int sig)
{
  int i, j;

  printf("\nsig_int starting\n");
  for (i = 0; i < 2000000; i ++)
      j += i*i;
  printf("\nsig_int finished\n");
  return;
}

/* Sleep function */
void sig_alrm(int sig)
{
  return;
}

unsigned int mysleep(unsigned int nsec)
{
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
     return(nsec);
  alarm(nsec);          /* start the timer */
  pause();              /* next caught signal wakes up */
```

```
  return(alarm(0));     /* turn of timer and return unslept seconds */
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от изпълнението на програмата.

```
$ a.out
            < въвеждаме <ctrl-c> преди да изтекат 4 сек. >

sig_int starting

sig_int finished
mysleep returned 3
$ a.out
            < изчакваме поне 4 сек. >
mysleep returned 0
```

Пример

Програмата 6.3 е пример за обработка на повтарящи се сигнали от един тип по един и същи начин, чрез една и съща потребителска функция. На всяка минута проверява дали през последната минута е осъществяван достъп до файл с име, зададено в аргумента, и ако е така извежда съобщение. Процесът може да бъде прекратен с клавишите <Ctrl>+<C> или с командата `kill`, т.е. чрез сигнал `SIGINT` или `SIGTERM`.

```
/* ------------------------------------------------------------ */
/* Catch SIGALRM, SIGINT, SIGTERM */

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

void wakeup(int);
void quit(int);

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

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

  if (stat(argv[1], &statbuf) == -1)
     err_sys_exit("stat error: %s", argv[1]);
  atime = statbuf.st_atime;
  signal(SIGINT, quit);
  signal(SIGTERM, quit);
  signal(SIGALRM, wakeup);
  for( ; ; ) {
     if (stat(argv[1], &statbuf) == -1)
        err_sys_exit("stat error: %s", argv[1]);
     if (atime != statbuf.st_atime) {
        printf("%s: accessed\n", argv[1]);
        atime = statbuf.st_atime; }
     alarm(60);         /* start the timer */
     pause();
  }
}

void quit(int sig)
{
  printf("Termination of process %d on signal %d\n", getpid(), sig);
```

```
  exit(0);
}

void wakeup(int sig)
{
  signal(SIGALRM, wakeup);  /* for safety in all version of UNIX systems */
}
/* ------------------------------------------------------------ */
```

Изпълняваме програмата във фонов режим. Чрез команда `cat` четем файла `file1`, който се следи от процеса. Работата на програмата прекратяваме с командата `kill`, чрез която изпращаме сигнал `SIGTERM` на процеса.

```
$ a.out file1 &
[1] 2234
$ cat file1
            < съдържанието на файл file1 >
file1: accessed
$ kill 2234
$ Termination of process 2234 on signal 15
$ a.out /no/such/file
stat error: /no/such/file: No such file or directory
$ a.out dir1 &
[1] 2239
$ ls dir1
            < списък на имената в каталога dir1 >
dir1: accessed
$ kill 2239
Termination of process 2239 on signal 15
```

Пример

Програма 6.4 илюстрира използването на сигналите при групи процеси. Процесът, в който се изпълнява програмата, създава два процеса-синове. Единият от тях изпълнява `setpgrp` и става лидер на нова група процеси, а другият остава в групата на процеса-баща. След това и двата сина изпълняват `pause` и се блокират до получаване на първи сигнал. Процесът-баща изчаква 5 секунди е изпраща сигнал `SIGINT` на всички процеси от групата си. Този сигнал ще убие единия от синовете му, който е в неговата група, но не и другия син, който е в друга група. Също така няма да убие и самия процес-баща, тъй като той игнорира сигнала, изпълнил е `signal`. След това бащата извежда съобщение с `printf` и изпълнява командата `ps`, която ни показва състоянието на процесите-синове. Другият процес-син може да бъде убит с командата `kill` (той е фонова група).

```
/* ------------------------------------------------------------ */
/* Signals and process groups */
#include <signal.h>
#include "ourhdr.h"
void wakeup(int);

main (void)
{
  int i;

  printf("Parent : pid=%d, group=%d\n", getpid(), getpgrp());
  for (i=0; i<2; i++) {
      if (fork() == 0) {   /* in child processes  */
         if (i & 01) setpgrp();
         printf("Child %d: pid=%d, group=%d\n", i, getpid(), getpgrp());
         pause();
```

```
        }
    }
    signal(SIGINT, SIG_IGN);    /* without signal kill also parent */
    signal(SIGALRM, wakeup);
    alarm(5);
    pause();
    kill(0, SIGINT);            /* send SIGINT to group */
    printf("Parent after kill\n");

    system("ps j");
    exit(0);
}

void wakeup(int sig)
{
    return;
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от изпълнението на програмата.

```
$ a.out
Parent : pid=16468, group=16468
Child 0: pid=16469, group=16468
Child 1: pid=16470, group=16470
Parent after kill
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     16468 Ss     501   0:00 -bash
12467 16468 16468 12467 tty1     16468 R+     501   0:00 a.out
16468 16469 16468 12467 tty1     16468 Z+     501   0:00 [a.out] <defunct>
16468 16470 16470 12467 tty1     16468 S      501   0:00 a.out
16468 16471 16468 12467 tty1     16468 R+     501   0:00 ps j
```

Като заключителен пример към управлението на процесите ще напишем програма, която представлява скелет на процес демон. Демоните играят важна роля в работата на операционната система, като управляват различни услуги. Най-яркият пример за демон е процеса `init`, който инициализира и поддържа йерархията на процесите. Други популярни демони са `crond`, `inetd`, `lpd` и др. Кои са характеристиките, които правят от един процес демон?

- Работи постоянно, като обикновено времето му на живот е от стартиране на системата до shutdown. Обикновено е син на процеса `init`.
- Чака настъпването на някакво събитие и тогава изпълнява услугата си.
- Няма управляващ терминал, което го защитава от сигналите, генерирани от терминала.
- Понякога създава друг(и) процес(и) - свое копие, който изпълнява отделна заявка за услуга.

Пример

Програма 6.5 е скелет на процес демон, който чака появата на определен файл. Името на файла се задава като аргумент. Когато файлът се появи нашият демон завършва.

```
/* ------------------------------------------------------------ */
/* Example daemon process */

#include <signal.h>
#include <sys/param.h>
#include <sys/stat.h>
#include "ourhdr.h"
```

```
void sig_alarm(int);

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

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

  printf("My daemon begins: pid=%d, ppid=%d, grp=%d, sid=%d\n",
        getpid(), getppid(), getpgrp(), getsid(0));

  if (getppid() != 1) {
     signal(SIGTSTP, SIG_IGN);
     signal(SIGTTOU, SIG_IGN);
     signal(SIGTTIN, SIG_IGN);
     if ((pid = fork()) < 0)
        err_sys_exit("cannot fork");
     if (pid > 0)
         exit(0); /* parent exits and parent of child becomes init */

/* child becomes session and group leader without tty */
     if(setsid() == -1)
        err_sys_exit("setsid error");
  }

  sleep(2);  /* wait the parent to exit */
  printf("My daemon is started: pid=%d, ppid=%d, grp=%d, sid=%d\n",
       getpid(), getppid(), getpgrp(), getsid(0));

  for (fd=0; fd < NOFILE; fd++)
     close(fd);
  errno = 0;

  chdir("/tmp");
  umask(0);

  signal(SIGALRM, sig_alarm);
  while (1)
  {

/* Do the daemon work */
     if(stat(argv[1], &statbuf) == 0){
        exit(0);
     }
     alarm(20);
     pause();
  }
}

void sig_alarm(int sig)
{
  signal(SIGALRM, sig_alarm);
  return;
}
/* ------------------------------------------------------------ */
```

Програмата изпълнява следните действията:

1. Игнорира сигналите, които се изпращат при вход/изход на терминала, когато процесът е пуснат във фонов режим (`SIGTSTP`, `SIGTTOU`, `SIGTTIN`). В противен случай, тези сигнали ще му повлияят. Ако първоначално процесът е син на `init`, не бива да се безпокоим за това.
2. Става син на процеса `init`.
3. Създава нова сесия и група в нея, на които е лидер. Това прекъсва и връзката на процеса с управляващия терминал.
4. Затваря всички файлови дескриптори.
5. Изменя текущия каталог на `/tmp` (или друг каталог, в който процесът работи).
6. Променя маската при създаване на файлове на 0.
7. Върти безкраен цикъл, в който „върши демонската си работа”.

Съобщения за работата си процес демон не може да извежда на стандартния изход, затова може да ги извежда в специален системен журнал чрез функцията `syslog`.

Следва примерен изход от изпълнението на програмата в Linux Kernel 2.6. Чрез командата `ps` можем да проверим съществуването на процеса.

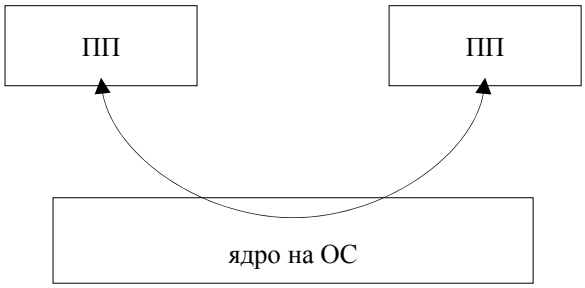
```
$ a.out /tmp/testf
My daemon begins: pid=16588, ppid=12467, grp=16588, sid=12467
$ My daemon is started: pid=16589, ppid=1, grp=16589, sid=16589
$ ps xj
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
2535 12467 12467 12467 tty1     16590 Ss     501   0:00 -bash
    1 16589 16589 16589 ?           -1 Ss     501   0:00 a.out /tmp/testf
12467 16590 16590 12467 tty1     16590 R+     501   0:00 ps xj
```

Когато по-късно създадем файл с име `/tmp/testf`, процесът завършва.

Седма глава

КОМУНИКАЦИИ МЕЖДУ ПРОЦЕСИ

IPC е съкращение от Interprocess Communication или комуникация между процеси. Когато два или повече конкурентни процеса взаимодействат помежду си, то между тях трябва да има съглашение за това и операционната система трябва да осигури някакъв механизъм за предаване на данни и синхронизация на работата им. Обикновено в едномашинна ОС информацията се предава през ядрото чрез някакъв механизъм за междупроцесна комуникация, както е показано на *Фиг.7.1*.



Фиг. 7.1. Комуникация между процеси в едномашинна ОС

С развитието на операционните системи от тип Unix и Linux в тях са включени различни методи и механизми за междупроцесна комуникация:

- програмни канали (pipes)
- именовани програмни канали (named pipes или FIFO файлове)
- съобщения (message queues)
- обща памет (shared memory)
- семафори (semaphores)

Когато два или повече процеса използват някакъв механизъм за обмен на информация, то IPC обекта трябва да има име или идентификатор. Така един от процесите ще създаде IPC обекта, а останалите ще получат достъп към този конкретен обект. Множеството от имена за определен вид IPC, наричаме пространство на имената. На *Фиг.7.2* са обобщени правилата за именуване на различните видове IPC, които ще разгледаме.

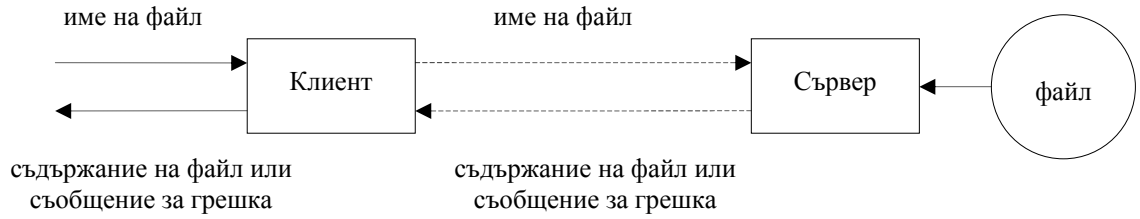
Вид IPC	Пространство на имената при създаване/отваряне	Идентификатор след отваряне
Програмен канал	без имена	файлов дескриптор
FIFO файл	име на файл	файлов дескриптор
Съобщения на System V	ключ от тип `key_t`	идентификатор на IPC в System V
Обща памет на System V	ключ от тип `key_t`	идентификатор на IPC в System V
Семафори на System V	ключ от тип `key_t`	идентификатор на IPC в System V

Ф*иг. 7.2*. Пространство на имената за различните IPC механизми

За илюстрация на различните механизми за междупроцесна комуникация ще използваме няколко класически примера. Един от тях е познатият ни Производител-Потребител. Друг модел за взаимодействащи процеси е Клиент-Сървер.

Примерът Клиент-Сървер, който ще реализираме, представлява файлов сървер и е показан на *Фиг.7.3*. Процесът-клиент чете от стандартния вход име на файл и го предава по IPC канала на сървера. Процесът-сървер чете името на файла от IPC канала и отваря файла за четене. При успех сървърът чете файла и го предава по IPC канала на клиента, иначе предава символен низ, съдържащ съобщение за грешка. Клиентът

приема по IPC канала и извежда полученото на стандартния изход. Пунктираните линии на *Фиг.7.3* съответстват на някаква форма на IPC, а плътните на входно/изходни операции с файл.



Ф*иг. 7.3*. Пример за Клиент-Сървър процеси

7.1. Програмни канали

Програмният канал е механизъм за комуникация между процеси, който осигурява еднопосочно предаване на неформатиран поток от данни (поток от байтове) между процесите и синхронизация на работата им. Реализират се два типа канали в различните версии на Unix и Linux системите:

- **неименован програмен канал** (**pipe**) - за комуникация между родствени процеси
- **именован програмен канал** (**named pipe** или **FIFO файл**) - за комуникация между независими процеси.

Като механизъм за комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове и има следните особености:

- За четене и писане в него се използват системните примитиви `read` и `write`, но дисциплината е FIFO.
- Каналът има доста ограничен капацитет, различен в различните реализации. В UNIX System V се ограничава от броя директни блокове в i-node, в Linux от размера на страница, или в по-новите версии е 65563. (Ограничението в `PIPE_BUF` от Програма 1.3 е максимален брой байта при една неделима операция писане в програмен канал).

Двата типа програмни канала се различават по начина, по който се създават и унищожават и по начина, по който процес първоначално осъществява достъп към канала. На *Фиг.7.4* са дадени функциите за основните операции при двата вида програмни канали.

	неименован	**FIFO файл**
създаване	`pipe`	`mknod`
отваряне		`open`
четене/писане	`read` и `write`	`read` и `write`
затваряне	`close`	`close`
унищожаване	автоматично при `close`	`unlink`

Фиг. 7.4. Операции над програмни канали

По-нататък ще разгледаме по-простия тип - неименован програмен канал, който се реализира още от най-ранните версии на Unix и за по-кратко ще го наричаме програмен канал.

Програмен канал се създава чрез примитива `pipe`.

```
#insert <unistd.h>
int pipe(int fd[2]);
                                                    Връща  0 при успех, -1 при грешка.
```

Създава се нов файл от тип програмен канал, което включва разпределяне и инициализиране на свободен индексен описател, както и при обикновените файлове, но за разлика от тях, каналът няма външно име и следователно не е част от йерархията на файловата система. След това каналът се отваря два пъти - един път за четене и един път за писане. Примитивът връща файлов дескриптор за четене в `fd[0]` и файлов дескриптор за писане в `fd[1]`.

Писане и четене в програмен канал се извършва с примитивите `write` и `read`, но достъпът до данните е с дисциплина FIFO, т.е. всяко писане е добавяне в края на файла и данните се четат от канала в реда, в който са записани. Това означава, че има някои особености в алгоритъма на примитивите `write` и `read`, когато първият аргумент е файлов дескриптор на програмен канал.

Писане в програмен канал

1. Ако в канала има достатъчно място, то данните се записват в края на файла. Увеличава се размера на файла със записания брой байта и се събуждат всички процеси, чакащи да четат от канала.

2. Ако в канала няма достатъчно място за всичките данни и броят байта, които се пишат при това извикване, е по-малък или равен на капацитета на канала, то ядрото блокира процеса. Когато бъде събуден от друг процес, изпълняващ `read`, той продължава както в случай 1.

3. Ако в канала няма достатъчно място за всичките данни, но броят байта, които се пишат при това извикване, е по-голям от капацитета на канала, то в канала се записват толкова байта, колкото е възможно и ядрото блокира процеса. Когато бъде събуден, той продължава да пише. В този случай операцията писане не е атомарна и е възможно състезание когато броят на процесите-писатели е по-голям от 1.

Четене от програмен канал

1. Ако в канала има някакви данни, то започва четене от началото на файла докато се удовлетвори искането на процеса или докато има данни в канала. Намалява се размера на файла с прочетения брой байта и се събуждат всички процеси, чакащи да пишат в канала.

2. Ако каналът е празен, ядрото блокира процеса. Когато бъде събуден от друг процес, изпълняващ `write`, той продължава както в случай 1.

Броят на процесите-четящи и процесите-пишещи в канала може да е различен и да е по-голям от 1, но тогава синхронизацията, осигурявана от механизма не е достатъчна.

Затваряне на програмен канал

Файловите дескриптори, върнати от `pipe`, за четене и писане в канал се освобождават с примитива `close` както и при работа с обикновени файлове, но има някои допълнения към алгоритъма на `close` при канали, чрез които се реализира синхронизацията на комуникиращите процеси и унищожаването на канала.

1. Ако при `close` се освободи последният файлов дескриптор за писане в канала (във всички процеси), то се събуждат всички процеси, чакащи да четат от канала, като `read` връща 0 (това означава край на файл).

2. Ако при `close` се освободи последният файлов дескриптор за четене от канала, то се събуждат всички процеси, чакащи да пишат в канала като им се изпраща сигнал `SIGPIPE`.

3. Когато се освободи и последният файлов дескриптор за работа с канала, програмният канал се унищожава.

За програмен канал не е разрешен примитива `lseek`.

Пример

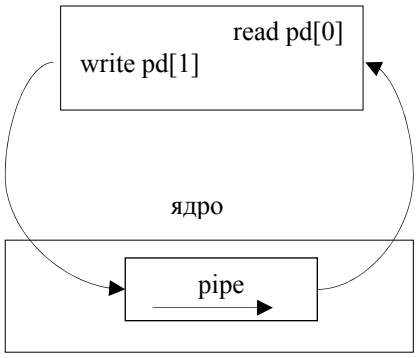
Програмният канал в един процес по схемата от *Фиг.7.5а* не е от голяма полза, но Програма 7.1 показва как се създава и използва програмен канал в един процес.

```
/* ------------------------------------------------------------ */
/* Simple example of pipe in one process */

#include "ourhdr.h"

main(void)
{
  int pd[2], n;
  char buff[MAXLINE];

  if (pipe(pd) < 0)
    err_sys_exit("pipe error");

  printf("read pipe fd=%d, write pipe fd=%d\n", pd[0], pd[1]);

  if (write(pd[1], "Hello World\n", 12) != 12)
    err_sys_exit("write error");

  if (( n = read(pd[0], buff, MAXLINE)) <= 0)
    err_sys_exit("read error");

  write(1, buff, n);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Изходът от тази програма е следния:

```
$ a.out
read pipe fd=3, write pipe fd=4
Hello Word
$ a.out > xx
$ cat xx
Hello Word
read pipe fd=3, write pipe fd=4
```

Забележете, че изходът от `write` при второто извикване, когато стандартният изход е пренасочен към файл, е преди този от `printf` въпреки, че в програмата са в обратен ред. Причината е, че в този случай изходът от `printf` се буферира (fully buffered) и не се извежда докато не се запълни буфера или процесът не завърши. Изходът от `printf`, когато стандартният вход е на терминала, се буферира по редове (line buffered), извежда се при символ за нов ред. Изходът при системен примитив `write` не се буферира.

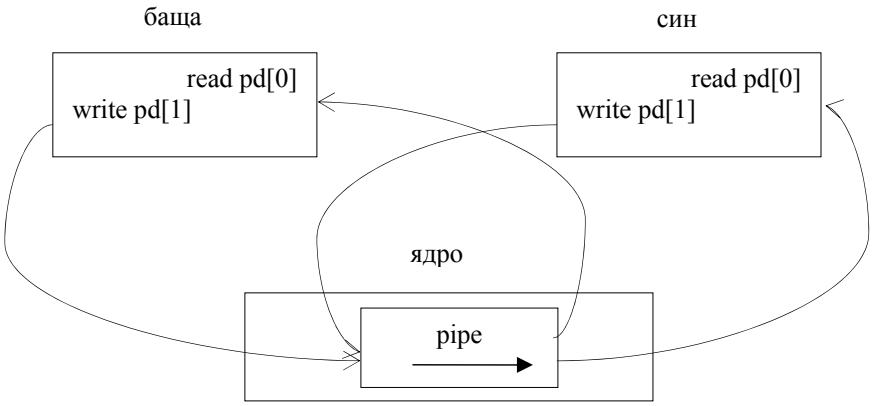


Фиг. 7.5а. Програмен канал в един процес веднага след `pipe`

Обикновено програмен канал се използва за комуникация между два процеса. Последователността от стъпки е следната:

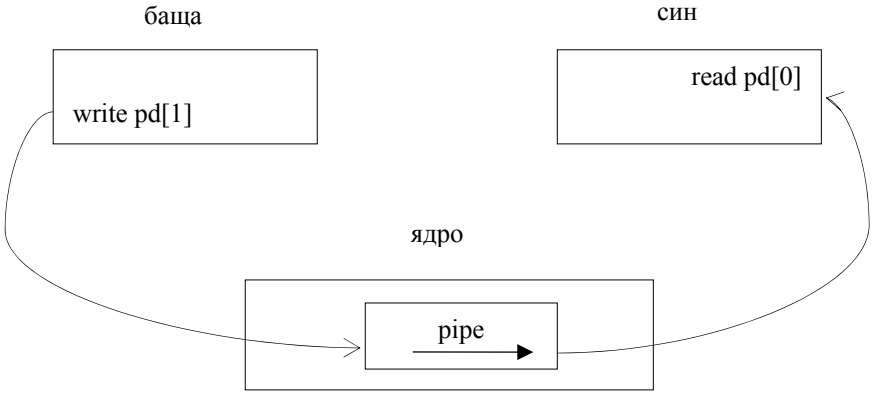
- процесът създава програмен канал - изпълнява `pipe`
- процесът създава нов процес - изпълнява `fork`

Резултатът от това е показан на следващата *Фиг.7.5б*. Има два процеса - баща и син, като синът е наследил от баща си двата файлови дескриптора за програмния канал.



Фиг. 7.5б. Програмен канал между два процеса веднага след `fork`

Ако след това бащата затвори файловия дескриптор за четене от програмния канал, а синът затвори този за писане (може и обратното), ще се получи еднопосочен канал за предаване на данни между два процеса - от бащата към сина, показан на *Фиг.7.5в*.



Фиг. 7.5в. Еднопосочна комуникация между два процеса с програмен канал

Пример

Програма 7.2 илюстрира механизма на програмните канали по схемата от Ф*иг.7.5в*.

```
/* ------------------------------------------------------------ */
/* Pipe from parent to child */

#include "ourhdr.h"

main(void)
{
  int pd[2], n;
  pid_t pid;
  char buff[MAXLINE];

  if (pipe(pd) < 0)
     err_sys_exit("pipe error");
  if ((pid = fork()) < 0)
     err_sys_exit("fork error");
  else if (pid > 0) {         /* parent */
      close(pd[0]);
      write(pd[1], "Hello World\n", 12);
  } else {                    /* child */
      close(pd[1]);
      n = read(pd[0], buff, MAXLINE);
      write(1, buff, n);
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Изходът от тази програма е:

```
$ a.out
Hello Word
```

Процесът, в който работи програмата, създава програмен канал след което създава и процес-син. Бащата пише в програмния канал, а синът чете от програмния канал и извежда прочетеното на стандартния изход.

И в двата примера ние четем от и пишем в програмния канал директно, т.е. като използваме файловите дескриптори върнати от `pipe`. По-интересно ще е да копираме съответните файлови дескритори, върнати от `pipe`, в стандартния вход или изход. След това процесът да извика за изпълнение друга програма чрез `exec`. Тази програма ще чете от стандартния вход или ще пише на стандартния изход, но в същност това ще е програмния канал, т.е. ще комуникира с друг процес. Това е често срещан начин за използване на програмните канали (при реализация на конвейер в shell), който е илюстриран в следващите два примера.

Пример

Програма 7.3 илюстрира механизма на програмните канали, като реализира конвейер на две програми, а по-точно "`ls | wc -l`".

```
/* ------------------------------------------------------------ */
/* Pipe Line ls | wc -l */

#include "ourhdr.h"
#define READ 0
#define WRITE 1
```

```
main (void)
{
  int pd[2], status;
  pid_t pid;

  if ((pid = fork()) < 0)
    err_sys_exit("fork error");
  if (pid == 0) {            /* in child */
     if (pipe(pd) < 0)
        err_sys_exit("pipe error");
     if ((pid = fork()) < 0)
        err_sys_exit("fork error");
     else if (pid == 0) {    /* in grandchild  */
      close(1);
      dup(pd[WRITE]);
      close(pd[READ]);
      close(pd[WRITE]);
      execlp("ls", "ls", 0);
      err_sys_exit("exec ls error");
     } else {                /* in child  */
      close(0);
      dup(pd[READ]);
      close(pd[READ]);
      close(pd[WRITE]);
      execlp("wc", "wc", "-l", 0);
      err_sys_exit("exec wc error");
      }
  }
  wait(&status);             /* in parent  */
  printf("Parent after end of pipe: status=%d\n", status);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида (в текущия каталог в момента има 11 файла).

```
$ a.out
11
Parent after end of pipe: status=0
```

Процесът, в който работи програмата, създава син, в който се изпълнява програмата `wc`. Той от своя страна също създава син за програмата `ls`. След това процесът, в който работи програмата, чака завършването на своя процес-син (`wc`)

Пример

Програма 7.4 показва друг вариант за реализация на конвейера "`ls | wc -l`". Двата процеса, в които работят програмите `ls` и `wc`, са синове на процеса за програмата. Процесът, в който работи програмата, чака завършването на втория си процес-син (`wc`).

```
/* ------------------------------------------------------------ */
/* Pipe Line ls | wc -l */

#include "ourhdr.h"
#define READ 0
#define WRITE 1

main (void)
{
  int pd[2], status;
```

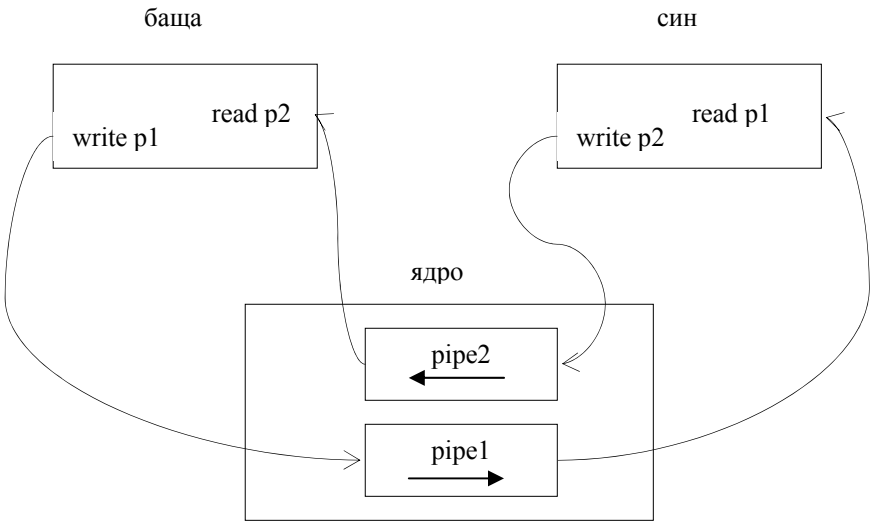
```
  pid_t pid;

  if (pipe(pd) < 0)
     err_sys_exit("pipe error");

  if ((pid = fork()) < 0)
     err_sys_exit("fork error for first child");

  if (pid == 0) {            /* in first child */
     close(1);
     dup(pd[WRITE]);
     close(pd[READ]);
     close(pd[WRITE]);
     execlp("ls", "ls", 0);
     err_sys_exit("exec ls error");
  }

  if ((pid = fork()) < 0)    /* in parent  */
     err_sys_exit("fork error for second child");

  if (pid == 0) {            /* in second child */
     close(0);
     dup(pd[READ]);
     close(pd[READ]);
     close(pd[WRITE]);
     execlp("wc", "wc", "-l", 0);
     err_sys_exit("exec wc error");
  }

  close(pd[READ]);           /* in parent  */
  close(pd[WRITE]);
  waitpid(pid, &status, 0);
  printf("Parent after end of pipe: status=%d\n", status);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим същия изход, както при Програма 7.3.

Ако е необходима двупосочна комуникация между два процеса, то трябва да се създадат два програмни канала, по един за всяко направление. Последователността от стъпки е следната:

- процесът създава `pipe1`
- процесът създава `pipe2`
- процесът създава нов процес – изпълнява `fork`
- бащата затваря файловия дескриптор за четене от `pipe1`
- бащата затваря файловия дескриптор за писане в `pipe2`
- синът затваря файловия дескриптор за писане в `pipe1`
- синът затваря файловия дескриптор за четене от `pipe2`

Така се получава схемата за комуникация показана на *Фиг. 7.6*.



Фиг. 7.6. Двупосочна комуникация между два процеса с програмни канали

Пример

Програма 7.5 реализира примера клиент-сървер чрез два програмни канала по схемата от *Фиг.7.6.* Клиентът работи в процеса-баща, а сървърът в процеса-син.

```
/* ------------------------------------------------------------ */
/* Client - Server with pipes */

#include <fcntl.h>
#include <sys/wait.h>
#include "ourhdr.h"
#define READ 0
#define WRITE 1

void client(int, int);
void server(int, int);

main(void)
{
  pid_t pid;
  int pd1[2], pd2[2];

  if (pipe(pd1) < 0 || pipe(pd2) < 0)
     err_sys_exit("pipe error");

  if ((pid = fork()) < 0)
     err_sys_exit("fork error");

  if (pid > 0)  {            /* parent - client */
     close(pd1[READ]);
     close(pd2[WRITE]);

     client(pd2[READ], pd1[WRITE]);

     waitpid(pid, NULL, 0);  /* parent wait for child */
     close(pd1[WRITE]);
     close(pd2[READ]);
     exit(0);
  } else {                   /* child - server */
     close(pd1[WRITE]);
     close(pd2[READ]);
```

```
     server(pd1[READ], pd2[WRITE]);

     close(pd1[READ]);
     close(pd2[WRITE]);
     exit(0);
  }
}

/*  Server  */
void server(int readfd, int writefd)
{
  char buff[MAXLINE];
  char errmsg[256];
  int  fd;
  ssize_t n;

/* read file name from IPC chanel */
  if ((n = read(readfd, buff, MAXLINE)) <= 0)
     err_sys_exit("server: filename read error");

  if ((fd = open(buff, O_RDONLY)) < 0) {
     sprintf(errmsg, ": can't open: %s\n", strerror(errno));
     strcat(buff, errmsg);
     n = strlen(buff);
     write(writefd, buff, n);

  } else {
/* file is open; read from file and write to IPC chanel */
     while ((n = read(fd, buff, MAXLINE)) > 0 )
         write(writefd, buff, n) ;

     close(fd);
  }
}

/*  Client  */
void client(int readfd, int writefd)
{
  char buff[MAXLINE];
  int n;

  printf("Type file name: ");
  fflush(stdout);
  if ((n = read(1, buff, MAXLINE)) == -1)
     err_sys_exit("client: filename read error");

  if( buff[n-1] == '\n' )
     n--;
  buff[n] = '\0';

  if (n == 0)
    err_exit("client: no file name");

/* write file name in IPC chanel */
  write(writefd, buff, n);

/* read from IPC chanel and write to stdout */
  while ((n = read(readfd, buff, MAXLINE)) > 0 )
      write(1, buff, n);
}
/* ------------------------------------------------------------ */
```

Ако изпълним програмата ще получим изход от вида.

```
$ cat file1
First line
Second line
$ a.out
Type file name: file1
First line
Second line
$ a.out
Type file name: /no/such/file
/no/such/file: can't open: No such file or directory
$ a.out
Type file name: /etc/shadow
/etc/shadow: can't open: Permission denied
```

7.2. IPC механизми на UNIX System V

IPC пакета на UNIX System V включва три механизма: съобщения, обща памет и семафори. Има доста общи черти при реализацията на тези три механизма. Затова ще започнем с разглеждане на общото между тях. На *Фиг.7.7* са дадени функциите за работа с трите IPC механизма.

	Съобщения	**Семафори**	**Обща памет**
Заглавен файл	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Създаване и отваряне	`msgget`	`semget`	`shmget`
Управление	`msgctl`	`semctl`	`shmctl`
IPC операции	`msgsnd, msgrcv`	`semop`	`shmat, shmdt`

Фиг. 7.7. Функции за работа с IPC механизми на UNIX System V

Когато се създава IPC обект в съответната функция `msgget`, `semget` или `shmget` се задава **ключ**. Ключът е от тип `key_t` и е цяло положително число. Той представлява външното име на IPC обекта (аналог на името на файл). При създаването на IPC обекта ядрото преобразува ключът в вътрешен **идентификатор**. Всеки IPC обект (и за трите механизма) се идентифицира в ядрото чрез този вътрешен идентификатор (аналог на файловия дескриптор при файлове). Този идентификатор се използва в примитивите за работа за съответния механизъм, например за изпращане или получаване на съобщение. Когато процес иска да получи достъп към вече създаден IPC обект (може и от друг процес), това отново става чрез съответната `XXXget` функция. Функциите `XXXget` са аналог на `creat` и `open` при файлове.

Трите функции `XXXget`, използвани за създаване и отваряне на IPC обект, приемат като първи аргумент *`key`* - ключа на обекта и имат още един общ аргумент *`flag`*. Този аргумент се конструира като побитово ИЛИ от флагове и битове, определящи правата на процесите (елемент `mode` в структурата `ipc_perm`). Възможните комбинации от флагове в аргумента *`flag`* и логиката на действие при функциите `XXXget` са обобщени във *Фиг.7.8*.

Аргумент *flag*	Не съществува обект за ключа *key*	Съществува обект за ключа *key*
няма флагове	грешка, errno=ENOENT	успех, отваря обекта
IPC_CREAT	успех, създава нов обект за *key*	успех, отваря обекта
IPC_CREAT|IPC_EXCL	успех, създава нов обект за *key*	грешка, errno=EEXIST

Фиг. 7.8. Логика на създаване и отваряне на IPC обект

Всеки създаден IPC обект се представя в ядрото чрез структура, съдържаща информация за него. Независимо от вида на IPC обекта тази структура включва елемент от тип ipc_perm, определен във файла <sys/ipc.h>. Структурата ipc_perm съдържа следните елементи.

```
struct ipc_perm {
      key_t key;              /* key */
      uid_t uid;              /* owner's user ID */
      gid_t gid;              /* owner's group ID */
      uid_t cuid;             /* creator's user ID */
      gid_t cgid;             /* creator's group ID */
      unsigned short mode;    /* access mode */
      unsigned short seq;     /* sequence number */
};
```

При създаване на IPC обект се инициализират всички елементи в тази структура.

Елементите uid, gid, cuid и cgid получават значения от ефективните идентификатори (euid, egid) на процеса, създаващ обекта.

Елементът mode се инициализира от значението в аргумента *flag* на функцията XXXget. Този елемент има същата структура и предназначение както кода на защита при файлове, т.е. определя правата на процесите за работа с IPC обекта. Разликата е, че тук се използват не всички битове, а само битовете r и w във всяка тройка на младшите 9 бита. Значението на битовете при различните механизми е показано на *Фиг.7.9*.

mode	**За кого е правото**	**Съобщения**	**Семафори**	**Обща памет**
0400	собственик	получаване	четене	четене
0200	собственик	изпращане	изменение	писане
0040	група	получаване	четене	четене
0020	група	изпращане	изменение	писане
0004	други	получаване	четене	четене
0002	други	изпращане	изменение	писане

Фиг. 7.9. Права на достъп до IPC обекти

По-късно чрез функциите XXXctl могат да бъдат изменени елементите uid, gid и mode. Така елементите uid и gid съдържат идентификаторите на текущия собственик на обекта. Елементите cuid и cgid не могат да бъдат изменяни, т.е. те винаги съдържат идентификаторите на създателя на обекта. Функциите XXXctl са аналог на chown и chmod при файлове.

Когато процес осъществява достъп до съществуващ IPC обект се извършва проверка на правата на два етапа. Първата проверка се извършва при извикване на съответната функция XXXget. В аргумента *flag* на функцията не трябва да има вдигнати битове, които не са вдигнати в елемента mode на структурата ipc_perm. Ако такива има, функцията завършва с грешка. Процесът може да пропусне тази проверка, като зададе 0 в аргумента *flag*. При всяка следваща операция над IPC обекта, напр.,

извикване на функции `msgsnd` или `msgrcv` за съобщения, се извършва отново проверка на правата на процеса в следната последователност.

1. Ако процесът принадлежи на администратора, той получава достъп.
2. Ако ефективният потребителски идентификатор на процеса е равен на `uid` или `cuid` на обекта, то ако съответният бит в елемента `mode` е вдигнат операцията се разрешава, иначе се отказва. (Напр., при `msgsnd` в този случай се гледа само бита 0200, а при `msgrcv` - бита 0400.)
3. Ако ефективният групов идентификатор на процеса е равен на `gid` или `cgid` на обекта, то ако съответният бит в елемента `mode` е вдигнат операцията се разрешава, иначе се отказва (бит 0020 или 0040).
4. Ако в предишните стъпки достъпът не е разрешен или отказан, то се проверяват битовете за другите в елемента `mode` (бит 0002 или 0004).

7.3. Съобщения

Един от методите за комуникация между процеси е чрез съобщения (Message passing). Процесите взаимодействат като си предават съобщения - един процес изпраща едно съобщение, а друг го получава. Съществуват различни логически модели на съобщения. Съобщенията в IPC пакета на UNIX System V са с:

- Косвена адресация

  Съобщения се предават в и получават от опашка на съобщенията (Message queue ще я съкращаваме на MQ).
- Автоматично буфериране

  Съществува системен буфер с ограничен капацитет, в който временно могат да се съхраняват изпратени и още неполучени съобщения.

Съществуват ограничения при всяка конкретна реализацията на този механизъм в различните Unix и Linux системи (извеждат се с командата `ipcs -lq`). В Linux Kernel 2.6.9 те са следните:

```
------ Messages: Limits --------
max queues system wide = 16
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

Всяка опашка на съобщенията се представя в ядрото чрез структура `msqid_ds`, определена в заглавния файл `<sys/msg.h>` и съдържаща следните елементи.

```
struct msqid_ds {
   struct ipc_perm msg_perm;
   time_t msg_stime;         /* time of last msgsnd */
   time_t msg_rtime;         /* time of last msgrcv */
   time_t msg_ctime;         /* time of last change */
   unsigned long msg_cbytes; /* current number of bytes on queue */
   msgqnum_t msg_qnum;       /* number of messages currently on queue */
   msglen_t msg_qbytes;      /* max number of bytes allowed on queue */
   pid_t msg_lspid;          /* pid of last msgsnd */
   pid_t msg_lrpid;          /* pid of last msgrcv */
};
```

Елементите на структурата съдържат информация за опашката: права на достъп и собственост (`msg_perm`), брой съобщения в опашката (`msg_qnum`) и обща дължина на всички съобщения в нея (`msg_cbytes`), време и идентификатор на процес, изпълнил последния `msgsnd` (`msg_stime`, `msg_lspid`) и същото за `msgrcv` (`msg_rtime`, `msg_lrpid`). Елементът `msg_qbytes` определя едно от ограниченията за съхранявани в опашката съобщения - максимална обща дължина на всички съобщения в опашката.

Опашка на съобщенията се създава или отваря чрез функцията `msgget`.

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

Връща идентификатор на MQ при успех, -1 при грешка.

Аргументът *`key`* съдържа ключа (външното име) на опашката. Нов обект MQ се създава, ако е изпълнено едно от условията:

- Не съществува MQ за ключа *`key`* и в аргумента *`flag`* е зададен флаг `IPC_CREAT`.
- В аргумента *`key`* е зададена константата `IPC_PRIVATE`.

В противен случай или се отваря съществуваща MQ или функцията завършва с грешка. При успех функцията връща вътрешния идентификатор на MQ, който се използва в останалите функции за идентифицирането ѝ.

След успешно изпълнение на `msgget`, в опашка с идентификатор *`msgid`* могат да се изпращат и получават съобщения чрез функциите `msgsnd` и `msgrcv`.

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msgid, struct msgbuf *msg, size_t size, int flag);
```

Връща 0 при успех, -1 при грешка.

```
int msgrcv(int msgid, struct msgbuf *msg, size_t size,
           long type, int flag);
```

Връща дължина на съобщението при успех, -1 при грешка.

По отношение на структурата на съобщенията има следните изисквания: всяко съобщение се състои от тип-цяло положително число и текст-масив от байтове с променлива дължина. Шаблонът за структура на едно съобщение е определен в заглавния файл `<sys/msg.h>`:

```
struct msgbuf {
     long mtype;      /* type of message */
     char mtext[1];   /* message text */
};
```

Използваме думата шаблон, защото в структурата `msgbuf` елементът за текст на съобщението `mtext` е определен с дължина 1, което в повечето случаи е недостатъчно. Най-често програмата трябва да определи своя структура за съобщението, в която елементът `mtext` да е с нужната дължина.

Всяко извикване на функцията `msgsnd` изпраща едно съобщение. Първият аргумент *`msgid`* е идентификатор на MQ. Аргументът *`msg`* е указател към изпращаното съобщение, а *`size`* е дължината на текста му (може да е и 0). Последният аргумент *`flag`* определя каква да е реакцията, ако съобщението не може да бъде изпратено веднага: има много съобщения в съответната опашка или въобще в системата.

- Ако *`flag`* съдържа 0, то процесът се блокира, до настъпването на едно от следните събития:
  - Освободи се достатъчно място за съобщението.
  - MQ с идентификатор *`msgid`* бъде унищожена от друг процес (код `EIDRM` в `errno`).

  - Процесът получи сигнал, който се обработва с потребителска функция (код `EINTR` в `errno`).

В първия случай съобщението се изпраща и функцията връща 0, а в другите два съобщението не е изпратено и функцията връща -1.

- Ако *`flag`* съдържа `IPC_NOWAIT`, то процесът не се блокира, а функцията завършва с грешка и код `EAGAIN` в `errno`.

За да получи едно съобщение процесът трябва да извика функцията `msgrcv`. Първите два аргумента са аналогични на аргументите в `msgsnd`. Аргументът *`size`* задава ограничение за максимален размер на чаканото съобщение (на текста му). Значението на аргумента *`type`* определя, кое от съобщенията в MQ ще бъде получено:

- Ако *`type`* `= 0` - първото съобщение в MQ.
- Ако *`type`* `> 0` - първото съобщение в MQ от тип *`type`*.
- Ако *`type`* `< 0` - първото съобщение в MQ от тип най-малкото число <= |*`type`*|.

По този начин чрез типа на съобщението и описаното действие на `msgrcv`, може да се реализират няколко потока за предаване на съобщения в рамките на една опашка на съобщенията, включително и двупосочна комуникация.

Аргументът *`flag`* в `msgrcv` представлява побитово ИЛИ от флагове, които определят какво да се прави ако в опашката няма съобщение от искания тип или има, но то е по-голямо от значението в аргумента *`size`*.

- Ако *`flag`* не съдържа `IPC_NOWAIT`, то процесът се блокира, до настъпването на едно от следните събития:
  - Получи се съобщение от чакания тип.
  - MQ с идентификатор *`msgid`* бъде унищожена от друг процес (код `EIDRM` в `errno`).
  - Процесът получи сигнал, който се обработва с потребителска функция (код `EINTR` в `errno`).
- Ако *`flag`* съдържа `IPC_NOWAIT`, то процесът не се блокира в случай, че няма съобщение от чакания тип, а функцията завършва с грешка и код `ENOMSG` в `errno`.
- Ако *`flag`* съдържа `MSG_NOERROR`, дългите съобщения се отрязват до размер *`size`*. В противен случай при дълго съобщение функцията връща грешка с код `E2BIG` в `errno`.

При успех функцията връща действителния размер на полученото съобщение (на текста му).

Функцията `msgctl` реализира операциите по управление на опашка на съобщения: получаване на информация за състоянието на MQ, промяна на някои атрибути, като правата на достъп и собственик, унищожаване на MQ.

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

Връща 0 при успех, -1 при грешка.

Аргументът *`msgid`* е идентификатор на опашката. Операцията се определя чрез аргумента *`cmd`*, който има следните значения:

`IPC_RMID` Унищожава се опашка с идентификатор *`msgid`*. Третият аргумент не се използва.

`IPC_STAT` Получава се информация за опашка с идентификатор *`msgid`* чрез аргумента *`buf`*.

`IPC_SET` Изменят се някои атрибути на опашка с идентификатор *`msgid`*. Могат да се изменят следните елементи на структурата `msqid_ds`: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, `msg_qbytes`. Аргументът *`buf`* определя новите значения.

За операцията `IPC_STAT` процесът трябва да има право за четене от опашката. При операциите `IPC_RMID` и `IPC_SET` процесът трябва да принадлежи на администратора или на създателя или на собственика на MQ. Увеличаването на `msg_qbytes` над ограничението `MSGMNB` (максимална обща дължина на съобщенията в една MQ) е позволено само на администратора.

IPC обект съществува докато не се унищожи явно чрез съответната функция `XXXctl`. Унищожаването на опашка на съобщения се извършва незабавно, т.е. при изпълнение на функцията `msgctl`. Ако има процеси, които са блокирани в `msgrcv` или `msgsnd`, те се събуждат и съответната функция връща -1 и код `EIDRM` в `errno`. Така е и при унищожаване на семафор, но не и при обща памет.

Пример

Програма 7.6 е прост пример, в който се създава опашка на съобщения, след това чрез функцията `msgctl` се получава информация за нея и накрая се унищожава.

```
/* ------------------------------------------------------------ */
/* Example of message queue */

#include <sys/ipc.h>
#include <sys/msg.h>
#include "ourhdr.h"
#define MODE 0600

struct msgb {
  int mtype;
  char mtext[1];
 };

main(void)
{
int msgid;
struct msgb msg;
struct msqid_ds info;

  if ((msgid = msgget(IPC_PRIVATE, MODE)) == -1)
     err_sys_exit("Msgget error");
  msg.mtype = 1;
  msg.mtext[0] = 'a';

  msgsnd(msgid, &msg, 1, 0);

  if (msgctl(msgid, IPC_STAT, &info) == -1)
       err_sys_exit("Msgctl IPC_STAT error");
  printf("MQ: %03o, cbytes=%lu, qnum=%lu, qbytes=%lu\n",
         info.msg_perm.mode&0777, info.msg_cbytes,
         info.msg_qnum, info.msg_qbytes);

  system("ipcs -q");
  if (msgctl(msgid, IPC_RMID, 0) == -1)
     err_sys_exit("Msgctl IPC_RMID error");
  printf("MQ destroyed\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата в Linux Kernel 2.0 получихме следния изход.

```
$ a.out
MQ: 600, cbytes=1, qnum=1, qbytes=16384

------ Message Queues --------
msqid     owner     perms     used-bytes  messages
128       moni      600       1           1

MQ destroyed
```

В Linux Kernel 2.6 получихме следния изход.

```
MQ: 600, cbytes=1, qnum=1, qbytes=16384

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000000 32768      moni       600        1            1

MQ destroyed
```

В примера най-напред се създава опашка на съобщенията и в нея се изпраща съобщение с размер 1 байт. След това се извиква `msgctl` с аргумент `IPC_STAT`, който връща информация за опашката, част от която извеждаме на стандартния изход с `printf`. Изпълняваме и командата `ipcs` за да сравним информацията. Вижда се, че изходът на командата `ipcs` е различен в двете версии на Linux. Накрая унищожаваме опашката.

Пример

Програма 7.7 реализира примера Производител-Потребител чрез съобщения. Производителят изпраща последователни цели числа на Потребителя, който ги умножава по две и извежда резултата на стандартния изход.

```
/* ------------------------------------------------------------ */
/* Producer - Consumer with message queue */

#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>
#include "ourhdr.h"

#define MSG_KEY 1234
#define MODE 0600

void cleanup(int);

struct msgb{
  long mtype;
  char mtext[MAXLINE];
};
static int mid;

main(void)
{
struct msgb msg;
int i, j, buf;

  if ((mid=msgget(MSG_KEY, IPC_CREAT|IPC_EXCL|MODE)) == -1)
     err_sys_exit("Msgget error");

  signal(SIGINT, cleanup);
```

```
  if (fork()) {         /* parent - consumer */
     signal(SIGTERM, cleanup);
     for (i=0; ; i++) {
       msgrcv(mid, (struct msgbuf *)&msg, MAXLINE,1,0);
       buf = *((int *)msg.mtext);
       buf *= 2;
       printf("Consumer: %d \n", buf);
     }
  } else {              /* child - producer */
     signal(SIGINT, SIG_DFL);
     for (i=0; ; i++) {
       for(j=0;j<10000000;j++); /* for some delay */
       msg.mtype = 1;
       *(int*)msg.mtext = i;
       msgsnd(mid, (struct msgbuf *)&msg, sizeof(int), 0);
     }
  }
}

/* Signal handler for SIGINT and SIGTERM */
void cleanup(int sig)
{
  if (msgctl(mid, IPC_RMID, 0) == -1)
     err_sys_exit("Msgctl IPC_RMID error");
  printf("MQ destroyed\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от изпълнението на програмата.

```
$ a.out
Consumer: 0
Consumer: 2
Consumer: 4
Consumer: 6
Consumer: 8
Consumer: 10
Consumer: 12
                        < въвеждаме <ctrl-c> >
MQ destroyed
```

Производителят работи в процеса-син, а потребителят в бащата, но биха могли да работят и в неродствени процеси. И двата процеса изпълняват безкрайни цикли, затова ги прекратяваме със сигнал `SIGINT` (с клавишите <Ctrl>+<C>) или `SIGTERM` (чрез команда `kill`). Всеки от тези сигнали убива процеса-производител. Процесът-потребител се грижи да унищожи опашката преди да завърши.

Пример

Следващите две програми реализират примера клиент-сървер чрез една опашка на съобщенията. За предаване на съобщениия от клиента към сървера ще използваме тип 1, а за обратното направление - тип 2. Клиентът и сървърът работят в два неродствени процеса. Програма 7.8а реализира сървера, а Програма 7.8б клиента. В двете програми се използват функциите `mesg_send` и `mesg_recv`.

```
/* ------------------------------------------------------------ */
int mesg_send(int id, Mesg *msgptr)
{
  return(msgsnd(id, (void *)&(msgptr->type), msgptr->len, 0) );
}
```

```
int mesg_recv(int id, Mesg *msgptr)
{
  int n;
  n = msgrcv(id, (void *)&(msgptr->type), MSGMAX, msgptr->type, 0);
  msgptr->len = n;

  return(n);       /* -1 - error, 0 - EOF, >0 */
}
/* ------------------------------------------------------------ */
```

Съобщенията, изпращани или получавани чрез тези функции, имат структура определена по следния начин.

```
typedef struct {
  int len;
  long type;
  char data[MSGMAX];
} Mesg;
```

Елементът `type` съдържа типа на съобщението, `len` - дължината на текста на съобщението и `data` - самия текст на съобщението.
Следва Програма 7.8a, реализираща сървера:

```
/* ------------------------------------------------------------ */
/* Server process with single message queue */

#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "ourhdr.h"

#define MSG_KEY  1234
#define MSG_MODE 0660

typedef struct {
  int len;
  long type;
  char data[MSGMAX];
} Mesg;
Mesg mesg;

int mesg_send(int, Mesg *);
int mesg_recv(int, Mesg *);
void server(int);

main(void)
{
  int mid;

/* Create the message queue */
  if ( (mid = msgget(MSG_KEY, IPC_CREAT|IPC_EXCL|MSG_MODE)) < 0 )
     err_sys_exit("server: can't get message queue: %d", MSG_KEY);

  server(mid);

  exit(0);
}

void server(int mid)
{
  char errmsg[256];
```

```
  int n, fd;

/* receive file name from client */
  mesg.type = 1;
  if (( n = mesg_recv(mid, &mesg)) <= 0) {
    mesg.type = 2;
    mesg.len = 0;
    mesg_send(mid, &mesg);
    err_exit("server: filename read error");
  }
  mesg.data[n] = '\0';
  mesg.type = 2;

/* open file */
  if ( (fd = open(mesg.data, O_RDONLY)) < 0 ) {
     sprintf(errmsg, ": can't open: %s\n",  strerror(errno));
     strcat(mesg.data, errmsg);
     mesg.len = strlen(mesg.data);
     mesg_send(mid, &mesg);
  } else {

/* file is open; read from file and send data to client */
      while (( n = read(fd, mesg.data, MSGMAX)) > 0 ) {
        mesg.len = n;
        mesg_send(mid, &mesg);
      }
      close(fd);
      if (n < 0)
        err_sys_ret("server: data read error");
  }
  mesg.len = 0;
  mesg_send(mid, &mesg);
}
/* ------------------------------------------------------------ */
```

Следва Програма 7.8б, реализираща клиента:

```
/* ------------------------------------------------------------ */
/* Client process with single message queue */

#include <sys/ipc.h>
#include <sys/msg.h>
#include "ourhdr.h"

#define MSG_KEY  1234

typedef struct {
  int len;
  long type;
  char data[MSGMAX];
} Mesg;

Mesg mesg;

int mesg_send(int, Mesg *);
int mesg_recv(int, Mesg *);
void client(int);

main(void)
{
  int mid;
```

```
/* Open the message queue */
  if (( mid = msgget(MSG_KEY, 0)) < 0 )
    err_sys_exit("client: can't get message queue: %d", MSG_KEY);

  client(mid);

/* Delete message queue */
  if (msgctl(mid, IPC_RMID, (struct msqid_ds *) 0)  < 0)
    err_sys_exit("client: can't RMID message queue: %d", MSG_KEY);
  exit(0);
}

void client(int mid)
{
  int n;
  printf("Type file name: ");
  fflush(stdout);
  if ((n = read(0, mesg.data, MSGMAX)) == -1)
     err_sys_exit("client: filename read error");

  if (mesg.data[n-1] == '\n')
    n--;
  if (n == 0)
     err_exit("client: no file name");

  mesg.data[n] = '\0';
  mesg.len = n;
  mesg.type = 1;
  mesg_send(mid, &mesg);     /* send file name to server */

/* receive data from server */
  mesg.type = 2;
  while (( n = mesg_recv(mid, &mesg)) > 0)

/* write received data to stdout */
     if (write(1, mesg.data, n) != n)
        err_exit("client: data write error");

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

За да изпълним примера трябва първо да извикаме сървера във фонов режим и след това клиента в привилегирован режим. Сървърът приема една заявка от клиент, изпълнява я и завършва. Когато клиентът получи съобщение с дължина 0 от сървера, приема това за край на връзката - унищожава опашката на съобщенията и завършва. Следва изходът от няколко изпълнения на програмите.

```
$ cat file1
First line
Second line
$ server_msg_si &
[1] 1664
$ client_msg_si
Type file name: file1
First line
Second line
[1]+  Done                    server_msg_si
$ server_msg_si &
[1] 1666
```

```
$ client_msg_si
Type file name: file_ivan
file_ivan: can't open: Permission denied
[1]+  Done                    server_msg_si
$ server_msg_si & client_msg_si
[1] 1668
Type file name: /no/such/file
/no/such/file: can't open: No such file or directory
[1]+  Done                    server_msg_si
```

Пример

Друг вариант за реализизация на примера клиент-сървер е когато един процес сървер изпълнява заявките на много процеси клиенти и съществува докато не бъде спрян. Комуникацията между сървера и клиентите може да се осъществи чрез една опашка на съобщенията. За предаване на съобщениия от клиента към сървера ще използваме тип 1, а за обратното направление - тип равен на pid на процеса клиент, за когото е съобщението. Тази реализация използва функциите `mesg_send` и `mesg_recv` от Пример 7.8, както и структурата за съобщение `Mesg`.
Следва Програма 7.8_2a, реализираща сървера:

```
/* ------------------------------------------------------------ */
/* Server process with single message queue and many clients */

#include <fcntl.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "ourhdr.h"

#define MSG_KEY   1234
#define MSG_MODE 0660
#define MSGMAX 8192

Mesg mesg;
int mid;

int mesg_send(int, Mesg *);
int mesg_recv(int, Mesg *);
void server_quit(int);
void server(int);

main(void)
{

/* Create the message queue */
  if ((mid = msgget(MSG_KEY, IPC_CREAT|IPC_EXCL|MSG_MODE)) < 0)
     err_sys_exit("server: can't get message queue: %d", MSG_KEY);

  signal(SIGTERM, server_quit);
  signal(SIGINT, server_quit);

  server(mid);

}

void server(int id)
{
  char errmsg[256];
  int n, fd;
  pid_t pid;
```

```
  char *ptr;

  for ( ; ; ) {
/* receive pid and file name from client */
     mesg.type = 1;
     if ((n = mesg_recv(id, &mesg)) <= 0) {
        err_ret("server: filename read error");
        continue;
    }
    mesg.data[n] = '\0';
    if ((ptr = (char *)strchr(mesg.data, ' ')) == NULL) {
        err_ret("server: error in request: %s", mesg.data);
        pid = atol(mesg.data);
        mesg.type = pid;
        sprintf(errmsg, ": error in request\n");
        strcat(mesg.data, errmsg);
        mesg.len = strlen(mesg.data);
        mesg_send(id, &mesg);    /* send error message to client */
        mesg.len = 0;
        mesg_send(id, &mesg);  /* send END of connection to client */
        continue;
    }

    *ptr++ = 0;
    pid = atol(mesg.data);
    mesg.type = pid;

    if ((fd = open(ptr, O_RDONLY)) < 0) {
       sprintf(errmsg, ": can't open: %s\n",  strerror(errno));
       strcat(mesg.data, errmsg);
       mesg.len = strlen(mesg.data);
       mesg_send(id, & mesg);    /* send error message to client */
    } else {
       while (( n = read(fd, mesg.data, MSGMAX)) > 0 ) {
         mesg.len = n;
         mesg_send(id, &mesg);   /* send data to client */
       }
       if (n < 0)
          err_sys_ret("server: data read error");
       close(fd);
    }
    mesg.len = 0;
    mesg_send(mid, &mesg);  /* send END of connection to client */
  }
}

/* Signal handler for SIGINT and SIGTERM */
void server_quit(int sig)
{
  msgctl(mid, IPC_RMID, 0);
  printf("server: is down\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва Програма 7.8_2б, реализираща клиента:

```
/* ------------------------------------------------------------ */
/* Client process with single message queue and many clients */

#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#include "ourhdr.h"

#define MSG_KEY   1234
#define MSGMAX 8192

Mesg mesg;

int mesg_send(int, Mesg *);
int mesg_recv(int, Mesg *);
void client(int);

main(void)
{
  int mid;

/* Open the message queue */
  if ((mid = msgget(MSG_KEY, 0)) < 0 )
    err_sys_exit("client: can't get message queue: %d", MSG_KEY);

  client(mid);
    exit(0);
}

void client(int id)
{
  int n;
  char *ptr;

  snprintf(mesg.data, MSGMAX, "%ld ", (long)getpid());
  n = strlen(mesg.data);
  ptr = mesg.data + n;

  printf("Type file name: ");
  fflush(stdout);
  if ((n=read(0, ptr, MSGMAX - n)) == -1)
     err_sys_exit("client: filename read error", NULL);
  if ( *(ptr+n-1) =='\n' )
      n--;
  if (n == 0)
     err_exit("client: no file name");
  *(ptr+n) = '\0';
  n = strlen(mesg.data);
  mesg.len = n;
  mesg.type = 1;
  mesg_send(id, &mesg);  /* send pid and file name to server */

/* receive data from server and write received data to stdout */
  mesg.type = getpid();
  while (( n = mesg_recv(id, &mesg)) > 0 )
     if (write(1, mesg.data, n) != n)
        err_exit("client: data write error");

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

За да изпълним примера трябва първо да извикаме сървера във фонов режим и след това клиентите в привилегирован режим. Сърверът изпълнява цикъл, в който приема заявка от клиент и я изпълнява. Съобщение с дължина 0, изпратено от сървера, се приема от клиента за край на връзката и той завършва. Когато сърверът получи сигнал

SIGTERM или SIGINT, той унищожава опашката на съобщенията и завършва. Следва изходът от няколко изпълнения на програмите.

```
$ server_msg_M &
[1] 1686
$ client_msg_M
Type file name: file1
First line
Second line
$ client_msg_M
Type file name: file_ivan
1688: can't open: Permission denied
$ client_msg_M
Type file name: /no/such/file
1689: can't open: No such file or directory
$ client_msg_M
Type file name: /bin
server: data read error: Is a directory
$ kill 1686
$ server: is down
```

7.4. Обща памет

Общата памет е най-бързият и прост метод за комуникция между процеси. Процесите взаимодействат като осъществяват достъп до една и съща област в паметта. Методът е бърз защото не изисква системни примитиви за предаване на данни, т.е. няма вход в ядрото и копиране на данни между потребителския процес и ядрото. Четенето и писането в общата памет е толкова бързо, колкото и достъпа до всяка една променлива на процеса.

Съществуват ограничения при всяка конкретна реализацията на този механизъм в различните Unix и Linux системи (извеждат се с командата `ipcs -lm`). В Linux Kernel 2.6.9 те са следните:

```
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1
```

Всеки сегмент обща памет (така се нарича IPC обект от този вид, ще съкращаваме на SHM) се представя в ядрото чрез структура `shmid_ds`, определена в заглавния файл `<sys/shm.h>` и съдържаща следните елементи.

```
struct shmid_ds {
      struct ipc_perm shm_perm;     /* operation perms */
      size_t shm_segsz;             /* size of segment (bytes) */
      time_t shm_atime;             /* last attach time */
      time_t shm_dtime;             /* last detach time */
      time_t shm_ctime;             /* last change time */
      pid_t shm_cpid;               /* pid of creator */
      pid_t shm_lpid;               /* pid of last operator */
      short shm_nattch;             /* no. of current attaches */
};
```

Елементите на структурата съдържат информация за SHM: права на достъп и собственост (`shm_perm`), размер на общата памет (`shm_segsz`), времена на последни операции над SHM (`shm_atime`, `shm_dtime`, `shm_ctime`), брой процеси, присъединили SHM към адресното си пространство (`shm_nattch`) и идентификатори на процеси, изпълнили операции над SHM (`shm_cpid`, `shm_lpid`).

Сегмент обща памет се създава или отваря чрез функцията `shmget`.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Връща идентификатор на SHM при успех, -1 при грешка.

Аргументите *`key`* и *`flag`* имат същото значение и логика на взаимодействие както при останалите функции `XXXget`. Аргументът *`size`* задава размера на създавания сегмент обща памет. Когато процесът не създава нов SHM, а получава достъп до съществуващ, аргументите *`size`* и *`flag`* могат да са 0. При отваряне на съществуващ сегмент се прави проверка дали не е отбелязан за унищожаване и ако да, примитивът завършва с грешка и код `EIDRM` в `errno`. При успех функцията връща вътрешен идентификатор, който се използва в останалите функции за идентифициране на SHM.

След успешно изпълнение на `shmget`, процесът първо трябва да присъедини SHM към адресното си пространство чрез функцията `shmat` (attach). След това вече може да чете и пише в общата памет както в собственото си адресно пространство. Когато не е необходим, SHM се освобождава чрез функцията `shmdt` (detach).

```
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, void *addr, int flag);
```

Връща адрес на SHM в процеса при успех, -1 при грешка.

```
int shmdt(void *addr);
```

Връща 0 при успех, -1 при грешка.

Функцията `shmat` присъединява SHM с идентификатор *`shmid`* към адресното пространство на процеса. Адресът на присъединяване се определя от аргументите *`addr`* и *`flag`*. Ако аргументът *`addr`* е 0, то ядрото определя адреса (това е препоръчваният начин). Ако аргументът *`addr`* не е 0, той трябва да задава виртуалния адрес на присъединяване. Като ако е вдигнат флаг `SHM_RND` в аргумента *`flag`*, значението в *`addr`* се подравнява на определяна от ядрото граница, а в противен случай се използва точно адресът, зададен в аргумента *`addr`*.

Ако в аргумента *`flag`* е вдигнат флаг `SHM_RDONLY`, сегментът се присъединява само за четене. В противен случай, се присъединява за четене и писане. Процесът трябва да има съответното право за SHM. При успех функцията връща действителния виртуален адрес на SHM в процеса.

Функцията `shmdt` освобождава SHM, присъединен преди това от процеса на адрес *`addr`*, от адресното пространство на процеса. Като аргумент на функцията се задава адресът на сегмента обща памет, а не идентификаторът *`shmid`*. Причината за това е, че един SHM може да бъде присъединен няколко пъти към адресното пространство на един процес на различни виртуални адреси.

Функцията `shmctl` реализира операциите по управление на общата памет: получаване на информация за състоянието на SHM, промяна на някои атрибути, като правата на достъп и собственик и унищожаване на SHM.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Връща 0 при успех, -1 при грешка.

Операциите в аргумента *cmd* са същите както при съобщенията: `IPC_STAT`, `IPC_SET`, `IPC_RMID`. Изискванията за правата на процеса са както при съобщения. При операция `IPC_RMID`, сегментът се отбелязва за унищожаване, но действителното освобождаване на паметта се извършва при изпълнение на `shmdt` от последния процес присъединил сегмента преди това (когато елементът `shm_nattch` в структура `shmid_ds` стане 0).

След `fork` процес-син наследява от бащата всички присъединени SHM. След `exec` всички присъединени SHM се освобождават (не унищожават). При завършване на процес (`exit`) присъединените SHM автоматично се освобождават.

Пример

Програма 7.9 е прост пример, в който се създава сегмент обща памет.

```
/* ------------------------------------------------------------ */
/* Create Shared Memory */

#include <sys/ipc.h>
#include <sys/shm.h>
#include"ourhdr.h"

#define SHM_SIZE 1024
#define SHM_MODE 0600

main(int argc, char *argv[])
{
  int shmid;
  key_t key;
  char *shm_adr;
  struct shmid_ds shm_buf;
  int shm_size;

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

/* Create a shared memory segment */
  key = atoi(argv[1]);
  shmid = shmget (key, SHM_SIZE, IPC_CREAT|IPC_EXCL|SHM_MODE);
  if (shmid == -1)
     err_sys_exit("shmget error");

/* Attach the shared memory segment */
  shm_adr = (char*)shmat (shmid, 0, 0);
  if (shm_adr == (void *)-1)
     err_sys_exit("shmat error");

/* Determine the size of shared memory segment */
  if (shmctl(shmid, IPC_STAT, &shm_buf) == -1)
     err_sys_exit("shmctl error");
  shm_size = shm_buf.shm_segsz;
  printf ("SHM attached at address: %p\nSegment size: %d\n",
          shm_adr, shm_size);

/* Write a string to the shared memory segment */
  sprintf (shm_adr, " Hello World");
```

```
/* Detach the shared memory segment */
  shmdt (shm_adr);

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

Като изпълнихме програмата в Linux Kernel 2.0 получихме следния изход.

```
$ a.out 1234
SHM attached at address: 0x40007000
Segment size: 1024
$ ipcs -m

------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch    status
0         moni      600       1024      0
```

В Linux Kernel 2.6 получихме следния изход.

```
$ a.out 1234
SHM attached at address: 0xb7fff000
Segment size: 1024
$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000004d2 950273     moni      600        1024       0
```

В примера най-напред се създава сегмент обща памет с ключ, зададен като аргумент на командния ред. След това сегментът се присъединява, в него се записва низа „Hello World” и процесът завършва. Когато след завършването на процеса изпълним командата `ipcs` виждаме, че сегментът съществува. (Изходът от `ipcs` в двете версии е различен.)

Пример

Програма 7.10 осъществява достъп до създадения в Програма 7.9 сегмент обща памет.

```
/* ------------------------------------------------------------ */
/* Read from Shared Memory and remove SHM */

#include <sys/ipc.h>
#include <sys/shm.h>
#include"ourhdr.h"

main(int argc, char *argv[])
{
  key_t key;
  int shmid;
  char *shm_adr;

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

/* Open a shared memory segment */
  key = atoi(argv[1]);
  shmid = shmget (key, 0, 0);
  if (shmid == -1)
     err_sys_exit("shmget error");
```

```
/* Attach the shared memory segment */
  shm_adr = (char*)shmat (shmid, 0, SHM_RDONLY);
  if (shm_adr == (void *)-1)
     err_sys_exit("shmat error");

/* Read from SHM and print the string */
printf ("%s\n", shm_adr);

/* Detach the shared memory segment */
  shmdt(shm_adr);

/* Destroy SHM */
  if (shmctl(shmid, IPC_RMID, 0) == -1)
     err_sys_exit("shmctl error");

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

Като изпълнихме програмата получихме следния изход.

```
$ a.out 1234
Hello World
```

В програмата най-напред се изпълнява `shmget` за съществуващия сегмент обща памет с ключ 1234, зададен като аргумент в командния ред. След това сегментът се присъединява и на стандартния изход се извежда низа, съдържащ се в сегмента (записан преди това при изпълнението на Програма 7.9). Накрая сегментът се унищожава. Това може да се провери с командата `ipcs`.

Пример

Пример 7.11 демонстрира състезание между два процеса, които четат и пишат в сегмент обща памет. В единия процес работи програмата `race1`, която създава сегмента обща памет и инициализира променливата `race`. В другия процес работи програмата `race2`, която отваря и присъединява създадения от `race1` сегмент и чете и пише в него.

Следва текстът на програмата `race1`.

```
/* ------------------------------------------------------------ */
/* Race condition with shared memory - process race1 */

#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>
#include"ourhdr.h"

#define SHM_SIZE 1024
#define SHM_MODE 0600
#define SHM_KEY 76
#define SHARED ptr-> race

struct shared {
   int race;
} *ptr;
int shmid;
void quit(int);
```

```
main(void)
{
  if ((shmid = shmget(SHM_KEY,SHM_SIZE,IPC_CREAT|IPC_EXCL|SHM_MODE))==-1)
      err_sys_exit("race1: shmget failed");

  ptr = (struct shared *)shmat(shmid, 0, 0);
  if ((ptr == (void *)-1)
     err_sys_exit("race1: shmat failed");

  printf ("Race1: shmid=%d\n", shmid);
  signal(SIGTERM, quit);

  while(1) {
    SHARED = 1;
    SHARED = 0;
  }
}

/* Signal handler for SIGTERM */
void quit(int sig)
{
  if (shmctl(shmid, IPC_RMID, 0) == -1)
     err_sys_exit("race1: shmctl failed");
  printf("Race1 quits\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва текстът на програмата `race2`.

```
/* ------------------------------------------------------------ */
/* Race condition with shared memory - process race2 */
#include <sys/ipc.h>
#include <sys/shm.h>
#include"ourhdr.h"

#define SHM_KEY 76
#define SHARED ptr-> race

struct shared {
   int race;
} *ptr;

main(void)
{
  int shmid;
  long i;
  int cntrace = 0;

  if((shmid = shmget(SHM_KEY, 0, 0)) == -1 )
      err_sys_exit("race2: shmget failed");

  ptr = (struct shared *)shmat(shmid, 0, SHM_RDONLY);
  if (ptr == (void *)-1)
     err_sys_exit("race2: shmat failed");

  printf ("Race2: shmid=%d\n", shmid);
  for (i=1; i<=50000000; i++ )
  if ( 2*SHARED != SHARED + SHARED ) cntrace++;

  printf("Race2: cntrace = %d\n", cntrace);
}
/* ------------------------------------------------------------ */
```

За да изпълним примера трябва първо да извикаме програмата `race1` във фонов режим и след това `race2`. Може да изпълняваме програмата `race2` няколко пъти и всеки път може да получаваме различно значение в променливата `cntrace`. Тя брои състезанията между двата процеса при достъп до общата променлива `race`. Процесът `race1` се прекратява с команда `kill`, която изпраща сигнал `SIGTERM`. Този сигнал се обработва от процеса с функцията `quit`, която унищожава сегмента обща памет и завършва процеса.

```
$ race1 & race2
[1] 431
Race1: shmid=1048577
Race2: shmid=1048577
Race2: cntrace = 4
$ race2
Race2: shmid=1048577
Race2: cntrace = 6
$ race2
Race2: shmid=1048577
Race2: cntrace = 3
$ kill 431              < прекратява процеса за race1 , след като race2 завърши >
$ Race1 quits
[1]+  Done                    race1
```

Проблемът при използването на обща памет за комуникация между процеси (демонстриран в Пример 7.11) е, че ядрото не синхронизира достъпа на процесите до общата памет, което води до състезания или други проблеми. Затова при метода за комуникация с обща памет трябва да се използва допълнителен механизъм за синхронизация. Такъв механизъм може да са семафорите.

7.5. Семафори

Семафорите са предложени от Дейкстра като механизам за осигуряване на взаимно изключване и синхронизация на процеси, използващи общи ресурси. Семафорите от IPC пакета на UNIX System V са по-сложни от тези на Дейкстра и имат следните особености:

- Семафорите се създават на масиви от един или повече семафора. Всеки елемент на масива има собствен брояч, който може да приема цели неотрицателни значения.
- Всеки масив семафори (понякога ще го наричаме за простота семафор) има ключ и идентификатор и се представя в ядрото чрез структура `semid_ds`, определена в заглавния файл `<sys/sem.h>`.
- Една операция може да се изпълни върху няколко елемента на масива семафори, т.е. да включва проверка и изменение на няколко брояча в масива семафори, като ядрото гарантира атомарността на цялата операция.
- Съществуват ограничения при всяка конкретна реализацията на този механизъм (извеждат се с командата `ipcs -ls`), които в Linux Kernel 2.6.9 са:

```
------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767
```

Структурата `semid_ds` съдържа следните елементи.

```
struct semid_ds {
  struct ipc_perm sem_perm;        /* permissions .. see ipc.h */
  time_t          sem_otime;       /* last semop time */
  time_t          sem_ctime;       /* last change time */
  unsigned short  sem_nsems;       /* no. of semaphores in array */
  struct sem      *sem_base;       /* ptr to first semaphore in array */
};
```

Елементите на структурата съдържат информация за масива семафори: права на достъп и собственост (`sem_perm`), брой елементи в масива семафори (`sem_nsems`), времена на последни операции над семафора (`sem_otime`, `sem_ctime`). Значението на елемента `sem_base` е указател към масив от елементи от тип `sem` - по един за всеки семафор в масива. Структурата от тип `sem` е вътрешна за ядрото. Тя реализира един семафор в масива и съдържа елементите:

`semval`	Значение на семафора (брояч).
`sempid`	Идентификатор на процеса, изпълнил последната операция над семафора.
`semncnt`	Брой процеси, чакащи да се увеличи значението на семафора.
`semnzcnt`	Брой процеси, чакащи значението на семафора да стане 0.

Масив от семафори се създава или отваря чрез функцията `semget`.

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Връща идентификатор на масив семафори при успех, -1 при грешка.

Аргументите *`key`* и *`flag`* имат същото предназначение както при другите два IPC механизма. Аргументът *`nsems`* задава броя на елементите в масива семафори. Когато се отваря съществуващ семафор в този аргумент може да се зададе 0. (Не може да се променя броят на елементите в съществуващ масив семафори.) При успех функцията връща идентификатор на масива семафори, който се използва във функциите `semop` и `semctl`.

Когато с `semget` се създава нов масив семафори, какво може да се каже за значенията в `semval`, т.е. инициализират ли се броячите на семафора. За съжаление реализациите в различните Unix и Linux системи може да се различават. В някои реализации при създаване на семафор, значението в `semval` за всички елементи се инициализира с 0. В други това не се прави, т.е. значението там е случайно. Тогава инициализацията трябва да се направи чрез функцията `semctl` с команда `SETVAL` или `SETALL`. Това е един от основните недостатъци на тези семафори, защото създаването и инициализацията на семафор вече не е атомарна операция. В документацията на Linux Kernel 2.6 нищо не се говори за началната инициализация при създаването.

След инициализацията на семафора, над него се изпълняват операции чрез функцията `semop`.

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Връща 0 при успех, -1 при грешка.

Аргументът *semid* е идентификатор на масив семафори. Аргументът *sops* е указател към масив от структури sembuf, който съдържа *nsops* елемента. Всяка структура sembuf определя операция над един семафор от масива *semid*.

```
struct sembuf {
   unsigned short  sem_num;        /* semaphore index in array */
   short           sem_op;         /* semaphore operation */
   short           sem_flg;        /* operation flags */
};
```

Елементът sem_num задава номера на елемента от масива семафори (номерацията започва от 0). Елементът sem_flg съдържа флагове. Ще спомена IPC_NOWAIT и SEM_UNDO. Когато е вдигнат флаг SEM_UNDO ядрото осигурява отмяна на операцията при завършване на процеса. Елементът sem_op задава самата операция.

1. Ако sem_op е положително цяло число, то се прибавя към значението в semval за семафора sem_num и операцията завършва успешно. Процесът трябва да има право за изменение на масива семафори.

2. Ако sem_op е 0, възможностите са следните:

- Ако значението в semval за семафора е 0, операцията завършва успешно.
- Ако значението в semval е по-голямо от 0, процесът се блокира докато настъпи едно от следните събития:
  - Значението в semval стане 0. В този случай операцията е успешна.
  - Семафорът бъде унищожен.
  - Процесът получи сигнал, за който реакцията е потребителска функция.

В последните два случая операцията е неуспешна, което се докладва с код за грешка в errno. Процесът трябва да има право за четене на масива семафори.

3. Ако sem_op е отрицателно цяло число, възможностите са следните:

- Ако значението в semval е по-голямо или равно на |sem_op|, то новото значение на семафора се изчислява на semval-|sem_op| и операцията завършва успешно.
- В противен случай, процесът се блокира докато настъпи едно от събитията:
  - Значението в semval стане по-голямо или равно на |sem_op|. Тогава се изчислява новото значение на semval и операцията завършва успешно.
  - Семафорът бъде унищожен.
  - Процесът получи сигнал, за който реакцията е потребителска функция.

В последните два случая операцията е неуспешна. Процесът трябва да има право за изменение на масива семафори.

Когато е вдигнат флаг IPC_NOWAIT в sem_flg и при изпълнение на операцията се налага блокиране на процеса, операцията е неуспешна с код на грешка EAGAIN.

За да има успех при изпълнение на функцията semop, трябва всички операции в масива *sops* да са успешни, т.е. гарантира се атомарност на цялата група от операции.

Функцията semctl реализира операциите по управление на семафори: получаване на информация за състоянието на семафори; инициализация на брояча; промяна на някои атрибути като правата на достъп и собственик; унищожаване на масива семафори.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int semctl(int semid, int semnum, int cmd, union semun arg);
```

Връща число >= 0 при успех, -1 при грешка.

Аргументът *semid* е идентификатор на масив семафори. Аргументът *cmd* определя управляващата операция. Някои операции се отнасят до целия масив семафори *semid*, а други само до определен елемент на масива, указан чрез аргумента *semnum*. Аргументът *arg* се използва по различен начин от операциите и затова е определен като union:

```
union semun {
   int val;                    /* value for SETVAL */
   struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
   unsigned short *array;      /* array for GETALL & SETALL */
};
```

Основните операциите в аргумента *cmd* са следните:

IPC_RMID Унищожава се масив семафори с идентификатор *semid*. Аргументите *semnum* и *arg* не се използват.

IPC_STAT Получава се информация за масива семафори с идентификатор *semid* чрез аргумента *arg.buf*. Аргументът *semnum* не се използва.

IPC_SET Изменят се някои атрибути на масива семафори с идентификатор *semid*. Могат да се изменят следните елементи на структурата semid_ds: sem_perm.uid, sem_perm.gid, sem_perm.mode. Аргументът *arg.buf* определя новите значения. Аргументът *semnum* не се използва.

GETALL Връща значенията на всички елементи в масива семафори чрез *arg.array*. Аргументът *semnum* не се използва.

GETVAL Връща значението на семафор с номер *semnum* като значение на функцията. Аргументът *arg* не се използва.

SETALL Променя значенията на всички семафори в масива, използвайки *arg.array*. Аргументът *semnum* не се използва.

SETVAL Променя значението на семафор с номер *semnum* на *arg.val*.

За операциите IPC_STAT, GETALL, GETVAL (и други GETxxx операции, които не са дадени тук), процесът трябва да има право за четене. При операциите IPC_RMID и IPC_STAT процесът трябва да принадлежи на администратора или на създателя или на собственика на масива семафори. Унищожаването на семафор се извършва незабавно, т.е. при изпълнение на функцията semctl. Ако има процеси, които са блокирани в semop, те се събуждат и функция връща -1 и код EIDRM в errno. За операциите SETALL и SETVAL процесът трябва да има право за изменение. Ако при тези операции значението на семафор да стане 0 или се увеличи и има блокирани процеси, които чакат някое от тези събития, те се събуждат.

Пример

Програма 7.12 е прост пример, в който се създава и инициализира един семафор.

```
/* ------------------------------------------------------------ */
/* Create semaphore */

#include <sys/ipc.h>
#include <sys/sem.h>
#include "ourhdr.h"
#define SEM_MODE 0600

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
};
```

```
main(int argc, char *argv[])
{
  int semid;
  int sem_value;
  union semun arg;
  key_t key;

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

/* Create a semaphore set */
  key = atoi(argv[1]);
  semid = semget(key, 1, IPC_CREAT|IPC_EXCL|SEM_MODE);
  if (semid == -1)
     if (errno == EEXIST) {
        printf("semget: key %d exists\n", key);
        semid = semget(key, 1, 0);
        if (semid == -1)
           err_sys_exit("semget error");
        else
           goto readval;
     }
     else
        err_sys_exit("semget error");

/* Initialize semval of semaphore */
  arg.val = 1;
  if ((semctl(semid, 0, SETVAL, arg.val)) == -1)
       err_sys_exit("semctl SETVAL error");

/* Read semval */
readval:
  if ((sem_value = semctl(semid, 0, GETVAL, 0)) == -1)
       err_sys_exit("semctl GETVAL error");
  printf("semval=%d\n", sem_value);
}
/* ------------------------------------------------------------ */
```

Като изпълнихме програмата в Linux Kernel 2.6 получихме следния изход.

```
$ a.out 123
semval=1
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0000007b 589832     moni       600        1

$ ipcs -s -i 589832

Semaphore Array semid=589832
uid=501      gid=500     cuid=501    cgid=500
mode=0600, access_perms=0600
nsems = 1
otime = Not set
ctime = Tue Jul 21 14:17:03 2009
semnum     value      ncount     zcount     pid
0          1          0          0          3032

$ a.out 123
semget: key 123 exists
```

```
semval=1
```

В програмата най-напред се създава масив от един семафор с ключ, зададен като аргумент в командния ред. След това семафорът се инициализира с 1, четем значението му и го извеждаме на стандартния изход. Когато, след завършването на процеса, изпълним командата `ipcs`, виждаме че семафорът съществува (чрез опциите `-s -i 589832` получаваме по-подробна информация). Можем да унищожим семафора с командата:

```
$ ipcrm -S 123    # или ipcrm -s 589832
```

Взаимно изключване на произволен брой процеси се реализира с един двоичен семафор, който приема значения 0 и 1. Според Дейкстра:

- Началното значение на семафора е 1.
- Значение 1 на семафора означава отворен.
- Значение 0 на семафора означава затворен.
- Операцията P(sem) блокира процеса, ако семафорът sem е 0 докато друг процес изпълни операция V(sem).

Такава семантика на двоичния семафор е реализирана в Програма 7.13.

Пример

Програма 7.13 е пример, в който се избягва състезанието на два процеса при достъп до общ брояч, съхраняван във файл, чрез един двоичен семафор.

```
/* ------------------------------------------------------------ */
/* Mutual exclusion with Dijkstra binary semaphore */

#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "ourhdr.h"
#define SEM_MODE 0600

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
};
int fd;

int sem_init(key_t, int);
void P(int);
void V(int);
int get_shared(void);
void put_shared(int);

main(int argc, char *argv[])
{
  key_t key;
  int semid;
  int loop;
  int i;
  int buff;

  if (argc < 4)
     err_exit("usage: a.out filename key loops");

  if ((fd = open(argv[1], O_CREAT|O_TRUNC|O_RDWR, 0600)) == -1)
     err_sys_exit("open error");
  put_shared(0);
```

```
/* Create and initialize a binary semaphore */
  key = atoi(argv[2]);
  if ((semid=sem_init(key, 1)) == -1)
     err_sys_exit("sem_init error");

  loop = atoi(argv[3]);
  if(fork()) {
     for (i=1; i<=loop; i++) {
         P(semid);
         buff = get_shared();
         buff += 1;
         put_shared(buff);
         V(semid);
      }
      wait(NULL);
      buff = get_shared();
      printf("Shared counter: %d \n", buff);
      semctl(semid, 0, IPC_RMID, 0); /* delete semaphore */
  }
  else {
      for (i=1; i<=loop; i++) {
          P(semid);
          buff = get_shared();
          buff += 2;
          put_shared(buff);
          V(semid);
       }
   }
}

/* Create and initialize a semaphore */
int sem_init(key_t key, int semval)
{
  int semid;
  union semun arg;

/* Create a semaphore */
  if ((semid=semget(key, 1, IPC_CREAT|IPC_EXCL|SEM_MODE)) == -1)
     return(-1);

/* Initialize the semaphore */
/* No need to initialize if semval = -1 */
  if (semval >= 0) {
     arg.val = semval;
     if ((semctl(semid, 0, SETVAL, arg.val)) == -1)
       return(-1);
  }
  return(semid);
}

/* P operation on semaphore */
void P(int semid)
{
  struct sembuf psem = {0, -1, SEM_UNDO};
  if (semop(semid, &psem, 1) == -1)
    err_sys_exit( "P(%d) failed", semid);
}
```

```
/* V operation on semaphore */
void V(int semid)
{
  struct sembuf vsem = {0,  1, SEM_UNDO};
  if (semop(semid, &vsem, 1) == -1)
     err_sys_exit("V(%d) failed", semid);
}

/* Write to shared file */
void put_shared(int i)
{
  lseek(fd, 0, SEEK_SET);
  write(fd, &i, sizeof(int));
}

/* Read from shared file */
int get_shared(void)
{
  int i;
  lseek(fd, 0, SEEK_SET);
  read(fd, &i, sizeof(int));
  return(i);
}
/* ------------------------------------------------------------ */
```

Изпълнихме програмата няколко пъти и получихме следния изход.

```
$ a.out testfile 123 400
Shared counter: 1200
$ a.out testfile 123 1000
Shared counter: 3000
$ a.out testfile 123 2000
Shared counter: 6000
$ a.out testfile 123 1000000
Shared counter: 3000000
$ ls -l testfile
-rw-------  1 moni staff 4 Sep 10 17:27 testfile
$ od -l testfile
0000000     3000000
0000004
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

В примера най-напред се създава масив от един семафор с ключ, зададен като втори аргумент на командния ред. След това семафорът се инициализира с 1. Процесът създава нов процес и двата процеса изпълняват цикли, в които четат и пишат в общия брояч, който се съхранява във файл. Накрая процесът-баща изчаква завършването на сина, извежда съдържанието на брояча от файла и унищожава семафора. Изпълняваме командите `ls` и `od` за да проверим съдържанието на файла `testfile`, а чрез командата `ipcs` се уверяваме, че семафорът е унищожен. Би могло да се напише вариант, в който броячът е в обща памет.

Друга възможна семантика на двоичен семафор е:

- Началното значение на семафора е 0.
- Значение 0 на семафора означава отворен.
- Значение 1 на семафора означава затворен.

Тогава ако при създаване на семафор с `semget` той се инициализира с 0, не се налага промяна на началното значение на семафора след създаването му, т.е. изпълнение на

`semctl` с команда `SETVAL`. Предимството на този начин е, че създаването и инициализацията на семафора се реализират чрез извикване на един примитив, следователно е атомарна операция. За създаване и инициализация може да се използва функцията от Пример 7.13, като се извика по следния начин: `sem_init(key, -1)`. Функциите за операциите `P` и `V` са показани в следващия пример.

Пример

Програма 7.14 е друга реализация на двоичен семафор с алтернативната семантика. Показани са само функциите за операциите `P` и `V`.

```
/* ------------------------------------------------------------ */
/* P operation on semaphore */
void P(int semid)
{
  struct sembuf psem[2] = {0, 0, 0,
                           0, 1, SEM_UNDO};

  if (semop(semid, psem, 2) == -1)
    err_sys_exit( "P(%d) failed", semid);
}

/* V operation on semaphore */
void V(int semid)
{
  struct sembuf vsem = {0,  -1, SEM_UNDO};

  if (semop(semid, &vsem, 1) == -1)
     err_sys_exit("V(%d) failed", semid);
}
/* ------------------------------------------------------------ */
```

Пример

Програма 7.15 реализира примера Производител-Потребител чрез обща памет и семафори. Моделът на взаимодействие е като в Програма 7.7.

```
/* ------------------------------------------------------------ */
/* Producer - Consumer with shared memory and semaphores */

#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "ourhdr.h"

#define MODE 0600
#define BUFS 20
#define SHM_SIZE BUFS*4 + 8
#define MUTEX 0
#define FULL  1
#define EMPTY 2

void cleanup(int);
int sem_init(key_t, unsigned short *);
int shm_init(key_t);
void P(int, int);
void V(int, int);
int removeitem(void);
void enteritem(int);

struct sh_buffer {
  int rp;
```

```
  int wp;
  int array[BUFS];
} *shptr;

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
};
int semid;
int shmid;

main(int argc, char *argv[])
{

  key_t key;
  int item, i;
  unsigned short semvals[] = {1, 0, BUFS};

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

  key = atoi(argv[1]);

/* Create and initialize shared memory */
  if ((shmid=shm_init(key)) == -1)
     err_sys_exit("shm_init error");

/* Create and initialize semaphores */
  if ((semid=sem_init(key, semvals)) == -1)
     err_sys_exit("sem_init error");

  signal(SIGINT, cleanup);
  if (fork()) {         /* parent - consumer  */
     signal(SIGTERM, cleanup);
     for (i=0; ; i++) {
       P(semid, FULL);
       P(semid, MUTEX);
         item = removeitem();
       V(semid, MUTEX);
       V(semid, EMPTY);
       item *= 2;
       printf("Consumer: %d \n", item);
     }

  } else {              /* child - producer */
     signal(SIGINT, SIG_DFL);
     for (item=0; ; item++) {
       P(semid, EMPTY);
       P(semid, MUTEX);
         enteritem(item);
       V(semid, MUTEX);
       V(semid, FULL);
       for(i=1; i<=1000000; i++); /* for some delay */
     }
  }
}

/* Create and initialize shared memory buffer */
int shm_init(key_t key)
{
```

```
  int shmid;
  int i;

  if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT|IPC_EXCL|MODE)) == -1)
      err_sys_exit("shmget failed");
  shptr = (struct sh_buffer* )shmat(shmid, 0, 0);
  if (shptr == (void *)-1)
     err_sys_exit("shmat failed");
  shptr->rp = 0;
  shptr->wp = 0;
  for (i=0; i<=BUFS-1; i++)
      shptr->array[i] = 0;
  return(shmid);
}

/* Create and initialize a semaphores */
int sem_init(key_t key, unsigned short *semvals)
{
  int semid;
  union semun arg;

/* Create a semaphores */
  if ((semid=semget(key, 3, IPC_CREAT|IPC_EXCL|MODE)) == -1)
     return(-1);
/* Initialize the semaphores */
     arg.array = semvals;
     if ((semctl(semid, 0, SETALL, arg.array)) == -1)
        return(-1);
  return(semid);
}

/* P operation on semaphore */
void P(int semid, int semnum)
{
  struct sembuf psem = {0, -1, SEM_UNDO};

  psem.sem_num = semnum;
  if (semop(semid, &psem, 1) == -1)
    err_sys_exit( "P(%d) failed", semnum);
}

/* V operation on semaphore */
void V(int semid, int semnum)
{
  struct sembuf vsem = {0,  1, SEM_UNDO};

  vsem.sem_num = semnum;
  if (semop(semid, &vsem, 1) == -1)
     err_sys_exit("V(%d) failed", semnum);
}

/* Write item to shared memory buffer */
void enteritem(int item)
{
  shptr->array[shptr->wp] = item;
  shptr->wp = (shptr->wp + 1)%BUFS;
  return;
}
```

```
/* Read item from shared memory buffer */
int removeitem(void)
{
  int item;
  item = shptr->array[shptr->rp];
  shptr->rp = (shptr->rp + 1)%BUFS;
  return(item);
}

/* Signal handler for SIGINT and SIGTERM */
void cleanup(int sig)
{
  if (semctl(semid, 0, IPC_RMID, 0) == -1)
     err_sys_exit("semctl IPC_RMID error");
  if (shmctl(shmid, IPC_RMID, 0) == -1)
     err_sys_exit("shmctl IPC_RMID error");
  printf("SHM and semaphores destroyed\n");
  exit(0);
}
/* ------------------------------------------------------------ */
```

Изпълнихме програмата и получихме следния изход.

```
$ a.out 123
Consumer: 0
Consumer: 2
Consumer: 4
Consumer: 6
Consumer: 8
Consumer: 10
Consumer: 12
                    < въвеждаме <ctrl-c>>
SHM and semaphores destroyed
```

За решаването на задачата Производител-Потребител с обща памет и семафори са необходими три семафора и общ буфер. В програмата се създава сегмент обща памет за буфера с ключ, зададен като аргумент на командния ред, който е с капацитет `BUFS` (20) елемента и е организиран като циклична опашка. След това се създава масив от три семафора със същия ключ. Семафорите се инициализират: първият (`MUTEX`) с 1, вторият (`FULL`) с 0 и третият (`EMPTY`) с `BUFS`. Процесът създава нов процес. Производителят работи в процеса-син, а потребителят в процеса-баща. И двата процеса изпълняват безкрайни цикли. Работата им може да се прекрати с клавишите < Ctrl>+<C> или с командата `kill`, при което процесът-производител завършва, а процесът-потребител унищожава сегмента обща памет и семафорите преди да завърши.

Осма глава

POSIX НИШКИ

Ще разгледаме нишките по стандарта POSIX 1003.1c, известни още като pthreads, които могат да се използват в съвременните версии на Unix и Linux системите. В някои версии, като UNIX System V, Solaris, Linux и др., нишките се реализират в ядрото. В други версии, като BSD, нишките не се поддържат от ядрото и се реализирата в потребителското пространство в една от библиотеките. Затова ще използваме термина функция, а не системен примитив. Функциите за работа с нишки, които ще разгледаме, осигуряват:

- Основни операции с нишки, като създаване, завършване и др.
- Механизми за синхронизация на работата на конкурентните нишки в един процес.

8.1. Основни операции с нишки

Първата - главна нишка във всеки процес се създава автоматично при създаване на процес. Друга нишка може да се създаде, когато една нишка изпълни функцията `pthread_create`.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Връща 0 при успех, число ≠ 0 при грешка.

Всяка нишка има идентификатор, който й се присвоява при създаването и е от тип `pthread_t`. При успех се връща идентификаторът на новата нишка чрез аргумента *`thread`*.

Аргументът *`attr`* определя някои характеристики на създаваната нишка или както ги наричат атрибути. Ако е зададено значение `NULL`, то атрибутите имат значения по премълчаване. Един от атрибутите определя типа на нишката: joinable или detached. Тип joinable означава, че друга нишка в процеса може да се синхронизира с момента на завършването й, т.е. след завършване на нишката тя не изчезва веднага (подобно на състояние зомби при процеси). Тип detached означава, че при завършване на нишката веднага се освобождават всички ресурси, заемани от нея, т.е. тя изчезва в момента на завършване. По премълчаване нишката се създава от тип joinable. Има и други атрибути, които определят дисциплината и параметри на планиране на нишки.

Когато нишката бъде създадена тя започва да изпълнява функцията *`start_routine`*, на която се предава аргумент *`arg`*. За разлика от процесите, където бащата и синът продължават изпълнението си след `fork` от една и съща точка, тук не е така, защото нишките на един процес имат общи ресурси. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Така създадената нишка завършва когато:

- Изпълни `return` от *`start_routine`*;
- Извика явно `pthread_exit`;
- Друга нишка я прекрати чрез `pthread_cancel`.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Аргументът *retval* е код на завършване, който нишката изработва. Той е предназначен за всяка друга нишка на процеса, която изпълни pthread_join. Но ако нишката е от тип detached, то след pthread_exit от нея не остава никаква следа, следователно и кодът не се съхранява. От тази функция няма връщане.

Чрез функцията pthread_join една нишка може да синхронизира изпълнението си със завършването на друга нишка на процеса (аналог е на примитива wait при процеси, но тук няма изискването текущата нишка да е баща на чаканата).

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

Връща 0 при успех, число ≠ 0 при грешка.

Аргументът *thread* е идентификатор на нишката, чието завършване се чака от текущата нишка. Текущата нишка се блокира докато не завърши нишка *thread*. Най-много една нишка може да изчака завършването на коя да е друга нишка, т.е. ако няколко нишки изпълнят pthread_join за една и съща нишка, вторият pthread_join ще върне грешка. При успех функцията връща 0 и чрез аргумента *value_ptr* се предава кодът на завършване на нишката *thread*. При грешка функцията връща код на грешка различен от 0.

Пример

Програма 8.1 илюстрира състезание между нишки. Двете нишки изпълняват една и съща функция, на която се предават аргументи чрез структура.

```
/* ------------------------------------------------------------ */
/*  Threads - Race condition with thteads */

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

/* Parameters to thread function */
struct pr_parms
{
  char ch;     /* The character to print */
  int count;   /* The number of times to print it */
};

int main (void)
{
  pthread_t th1;
  pthread_t th2;
  struct pr_parms th1_args;
  struct pr_parms th2_args;
  void *ch_print();

/* Create a thread to print 3000 x's */
  th1_args.ch = 'x';
  th1_args.count = 3000;
  pthread_create(&th1, NULL, ch_print, &th1_args);

/* Create a thread to print 2000 o's */
  th2_args.ch = 'o';
  th2_args.count = 2000;
  pthread_create(&th2, NULL, ch_print, &th2_args);
```

```
/* Wait the first thread to finish */
  pthread_join(th1, NULL);
/* Wait the second thread to finish */
  pthread_join(th2, NULL);

  exit(0);
}

/* Thread function */
void *ch_print (void *param)
{
  struct pr_parms* p = (struct pr_parms*)param;
  int i, j;
  for (i=0; i<p->count; ++i){
    fputc (p->ch, stdout);
    for(j=0; j<10000; j++);  /* for some delay */
  }
  pthread_exit((void *)0);
}
/* ---------------------------------------------------------- */
```

При компилиране на програма с нишки трябва да включим библиотеката `pthread`:

```
$ cc pthread_print.c -lpthread
```

Като изпълним програмата на стандартния изход получаваме изход, в който се редуват последователности от символа ‘o’ и последователности от символа ‘x’ с различни дължини.

8.2. Механизъм mutex

Mutex е механизъм за блокиране и деблокиране на нишки, чрез който може да се реализира взаимно изключване на нишки при достъп до общи променливи. Един обект mutex има две състояния:

- unlocked – свободен и не принадлежи на никоя нишка
- locked – заключен и принадлежи на нишката, която го е заключила.

Ако обект mutex е в състояние locked, той може да принадлежи само на една нишка. Следва описание на функциите за работа с обект mutex.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Връщат 0 при успех, число ≠ 0 при грешка.

Нов обект mutex се създава чрез функцията `pthread_mutex_init`. Идентификатор на новосъздадения обект mutex е променлива от тип `pthread_mutex_t` и се връща чрез аргумента *`mutex`*. Първоначално обектът mutex е в състояние unlocked. Вторият аргумент *`mutexattr`* задава атрибутите на създавания mutex. В Linux се реализира атрибут тип на mutex, който може да е: fast, recursive или error checking. Типът влияе на семантиката на последващите операции над обекта

mutex. По премълчаване, ако аргументът е `NULL`, се създава mutex от тип fast. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Mutex е общ ресурс, т.е. ресурс на процеса, а не на нишката, която го създава. Това означава, че не се унищожава или освобождава при завършване на нишката, която го е създала.

Чрез функцията `pthread_mutex_lock` текущата нишка се опитва да заключи обекта *`mutex`*. Възможните случаи при изпълнението са следните.

1. Ако *`mutex`* е свободен, то състоянието му се сменя в заключен от текущата нишка и тя продължава изпълнението си.
2. Ако *`mutex`* е заключен от някоя друга нишка, то текущата нишка бива блокирана, докато се смени състоянието му в unlocked от нишката, която го притежава в момента.
3. Ако *`mutex`* е заключен от текущата нишка, действието зависи от типа на *`mutex`*.
   - При тип fast текущата нишка се блокира (това може да доведе до дедлок).
   - При тип recursive се увеличава брояч (броят се многократните заключвания на един обект mutex от една нишка) и функцията завършва успешно.
   - При тип error checking това се счита за грешка.

С функцията `pthread_mutex_unlock` текущата нишка се опитва да освободи обекта *`mutex`*. Възможните случаи при изпълнението са:

1. Ако текущата нишка е настоящия притежател на обекта *`mutex`*, то се сменя състоянието му в unlocked. Ако има нишки, чакащи този обект *`mutex`* (блокирани в `pthread_mutex_lock`), то една от тях се събужда и й се дава възможност да се опита отново да получи *`mutex`*. Но ако *`mutex`* е от тип recursive се намалява броячът и когато той стане 0, тогава се прави отключване и събуждане.
2. Ако *`mutex`* е в състояние unlocked или е locked от друга нишка, то действието зависи от типа на *`mutex`*:
   - При тип fast не се прави проверка.
   - При тип error checking и recursive това е грешка.

При успех функциите връщат 0, а при грешка връщат различен от 0 код на грешка.

Функцията `pthread_mutex_destroy` унищожава обекта *`mutex`*, който трябва да е в състояние отключен иначе се счита за грешка. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Пример

Програма 8.2 илюстрира взаимно изключване при нишки чрез mutex.

```
/* ------------------------------------------------------------ */
/* Threads - Mutual Exclusion with mutex */

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

pthread_mutex_t mut;
int sum;

main(void)
{
  int ret, status, cnt1, cnt2;
  pthread_t th1, th2;
  void *race_func();

  cnt1 = 1;
  cnt2 = 2;
  sum = 0;
```

```
  ret = pthread_mutex_init(&mut, NULL);
  if (ret != 0) err_exit("Error in pthread_mutex_init");

  ret = pthread_create(&th1, NULL, race_func, &cnt1);
  if (ret != 0) err_exit("Error in pthread_create 1");

  ret = pthread_create(&th2, NULL, race_func, &cnt2);
  if (ret != 0) err_exit("Error in pthread_create 2");

  pthread_join(th1, &status);
//  printf("Thread 1 exit status: %d\n", status);

  pthread_join(th2, &status);
//  printf("Thread 2 exit status: %d\n", status);

  printf("\nSUM = %d\n", sum);
  exit(0);
}

/* Thread function */
void *race_func(void *ptr)
{
  int cnt, i, j;
  cnt = *(int *)ptr;
  printf("Thread %d\n", cnt);
  for (i=1; i<=100; i++) {
       pthread_mutex_lock(&mut);
       sum = sum + cnt;
       printf("%d\n", sum);
       pthread_mutex_unlock(&mut);
       for (j=0; j<400000; j++);    /*for some delay */
  }
  pthread_exit((void *)0);
}
/* ---------------------------------------------------------- */
```

Изпълнихме програмата и получихме следния изход.

```
$ a.out
Thread 1
1
2
3
Thread 2
5
7
8
9
10
11
12
14
16
18
                    <още много редове, в които се вижда как се редуват двете нишки>
300

SUM = 300
```

8.3. Механизъм condition

Condition е съкращение от condition variables или условни променливи. Това е друг механизъм за синхронизация, чрез който могат да се реализират по-сложни условия за блокиране и деблокиране на нишки. С всеки обект condition е свързано условие, при което нишката може да работи (или обратно, условие при което нишката ще се блокира). Това условие може да е произволно сложно. В следващия пример ще разгледаме едно много просто условие: ако променливата `flag` е 0 нишката ще се блокира, а ако `flag` е 1 ще работи.

Една нишка може да изпълнява операция *wait* върху обект condition, при което нишката се блокира докато друга нишка изпълни операция *signal* над същия обект condition. Когато една нишка изпълнява операция *signal*, то ако има нишки блокирани по обекта condition, се събужда една, в противен случай нищо не се прави. Операциите *wait* и *signal* са подобни на операциите P и V при семафори, но има една съществена разлика. Семафорът има брояч, който помни събуждането ако няма кого да буди, докато тук, ако няма блокирани нишки, събуждането не се помни.

Как трябва да се използва обект condition за реализиране на условна синхронизация? Следва описание на схемата на работа в случай на две нишки.

- В нишката, която трябва да чака условието за да работи:
  1. Проверява условието.
  2. Ако условието не е изпълнено, извиква операцията *wait*.
- В нишката, която променя и сигнализира условието:
  1. Променя променливите, от които зависи истинността на условието.
  2. Извиква операцията *signal.*

Всяка от двете нишки изпълнява две стъпки - проверка или изменение на условието и операция над обекта condition. За да няма състезание между тях е необходимо да се осигури атомарност на двете стъпки. Затова всеки обект condition се използва заедно с един обект mutex. Този mutex трябва да се заключва явно преди първата стъпка и да се отключва след втората, във всяка от двете нишки. Освен това той автоматично се отключва и заключва от операцията *wait*.

Следва описание на функциите за работа с обект condition.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *condattr);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Връщат 0 при успех, число ≠ 0 при грешка.

Нов обект се създава чрез функцията `pthread_cond_init`. Идентификатор на новосъздадения обект е променлива от тип `pthread_cond_t` и се връща чрез аргумента *`cond`*. Вторият аргумент *`condattr`* задава атрибутите на създавания обект condition. Ако в него е зададено значение `NULL`, атрибутите имат значения по

премълчаване. Реализацията в Linux не поддържа никакви атрибути, така че вторият аргумент се игнорира.

Чрез функцията `pthread_cond_signal` може да се събуди една нишка. Ако има нишки, блокирани в `pthread_cond_wait` по *`cond`*, се събужда една от тях. Ако няма блокирани нишки, нищо не се прави. Функцията `pthread_cond_broadcast` събужда всички нишки, блокирани по *`cond`*.

Нишка се блокира чрез функцията `pthread_cond_wait`. Текущата нишка се блокира по *`cond`*, докато друга нишка я събуди. Когато се извиква тази функция *`mutex`* трябва вече да е заключен от текущата нишка. Затова тази функция автоматично го отключва и блокира нишката. Когато по-късно бъде събудена от друга нишка, изпълняваща `pthread_cond_signal`, то *`mutex`* се заключва отново автоматично. След това нишката трябва отново да се провери условието, свързано с *`cond`* (първата стъпка от горната схема).

Функцията `pthread_cond_destroy` унищожава обекта *`cond`*, ако няма нишки блокирани по него.

При успех функциите връщат 0, а при грешка връщат различен от 0 код на грешка.

И така схемата на работа в двете нишки е следната.

- В нишката, която трябва да чака условието за да работи:

```
pthread_mutex_lock(&mut);
while( ! условие_за_да_работи )
   pthread_cond_wait(&cv, &mut);
pthread_mutex_unlock(&mut);
```

- В нишката, която променя и сигнализира условието:

```
pthread_mutex_lock(&mut);
изменя променливите, определящи истинността на условието;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mut);
```

Пример

Програма 8.3 илюстрира работа с обект condition, като реализира синхронизация на две нишки. Условието е много просто: ако променливата `flag` е 0 нишката, в която работи функцията `th_wait`, ще се блокира, а ако `flag` е 1 ще работи. Другата нишка, в която работи функцията `th_sign`, изменя условието, т.е. изменя променливата `flag`.

```
/* ------------------------------------------------------------ */
/*  Threads - Condition Variable */

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

pthread_mutex_t flag_mut;
pthread_cond_t flag_cv;
int flag;

main(int argc, char *argv[])
{
  int ret;
  pthread_t th1, th2;
  void *th_sign();
  void *th_wait();
  char *mess1 = "Hello";
  char *mess2 = "World";
```

```
  if (argc >= 3) {
      mess1 = argv[1];
      mess2 = argv[2]; }

/* Initialize the mutex and condition variables */
  ret = pthread_mutex_init(&flag_mut, NULL);
  if (ret != 0) err_exit("Error in pthread_mutex_init");

  pthread_cond_init(&flag_cv, NULL);

  flag = 0;

/* Create the threads */
  ret = pthread_create(&th1, NULL, th_wait, mess2);
  if (ret != 0) err_exit("Error in pthread_create 1");

  ret = pthread_create(&th2, NULL, th_sign, mess1);
  if (ret != 0) err_exit("Error in pthread_create 2");

/* Wait the threads to finish */
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  exit(0);
}

/* Signal Thread function */
void *th_sign(void *ptr)
{
  char *msg;
  msg = (char *)ptr;
  printf("Signal thread\n");
  printf("%s\n", msg);

/* Signal the condition: flag is set */
  pthread_mutex_lock(&flag_mut);
  flag = 1;
  pthread_cond_signal(&flag_cv);
  pthread_mutex_unlock(&flag_mut);

  pthread_exit((void *)0);
}

/* Wait Thread function */
void *th_wait(void *ptr)
{
  char *msg;
  msg = (char *)ptr;
  printf("Wait thread\n");

/* Wait the condition: flag is set */
  pthread_mutex_lock(&flag_mut);
  while( !flag )
       pthread_cond_wait(&flag_cv, &flag_mut);
  pthread_mutex_unlock(&flag_mut);

/* Do the work */
  printf("%s\n", msg);
  pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Изпълнихме програмата и получихме следния изход.

```
$ a.out
Wait thread
Signal thread
Hello
World
$ a.out first second
Wait thread
Signal thread
first
second
```

Пример

Програма 8.4 реализира по-сложната задача Производител-Потребител. Производителят и потребителят работят в две конкурентни нишки. За синхронизацията им са необходими два обект condition - `cv_empty` и `cv_full`. Условието, свързано с `cv_full`, е: ако буферът е пълен (променливата `count` е равна на размера на буфера) нишката-производител ще се блокира. Условието, свързано с `cv_empty`, е: ако буфера е празен (променливата `count` е равна на 0) нишката-потребител ще се блокира. Следователно, нишката-производител се блокира по `cv_full` и сигнализира `cv_empty`, а нишката-потребител се блокира по `cv_empty` и сигнализира `cv_full`.

```
/* ------------------------------------------------------------ */
/*  Threads - Produces-Consumer with Condition Variables */

#include <pthread.h>
#include "ourhdr.h"
#define BUFS 20

pthread_mutex_t mut;
pthread_cond_t full, empty;
int count;
int buffer[BUFS];
int rp, wp;
void enter_item();
int remove_item();

main(int argc, char *argv[])
{
  int ret;
  int loop, i;
  pthread_t th1, th2;
  void *th_prod();
  void *th_cons();

  if (argc < 2)
     err_exit("usage: a.out loop");
  loop = atoi(argv[1]);

/* Initialize the mutex and condition variables */

   ret = pthread_mutex_init(&mut, NULL);
   if (ret != 0) err_exit("Error in pthread_mutex_init");

   pthread_cond_init(&empty, NULL);
   pthread_cond_init(&full, NULL);

/* Initialize the shared variables */
   rp = 0;
```

```
   wp = 0;
   count = 0;
   for (i=0; i<=BUFS-1; i++)
       buffer[i] = 0;

/* Create the threads */
   ret = pthread_create(&th1, NULL, th_prod, &loop);
   if (ret != 0) err_exit("Error in pthread_create 1");

   ret = pthread_create(&th2, NULL, th_cons, &loop);
   if (ret != 0) err_exit("Error in pthread_create 2");

/* Wait the threads to finish */
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   exit(0);
}

/* Producer Thread function */
void *th_prod(void *ptr)
{
int item;
int loop;

   loop = *(int *)ptr;

   for (item=1; item<=loop; item++) {
       enter_item(item);
  }
   pthread_exit((void *)0);
}

/* Consumer Thread function */
void *th_cons(void *ptr)
{
int item, sum;
int loop, i;
   sum = 0;
   loop = *(int *)ptr;
   for (i=1; i<=loop; i++) {
        item = remove_item();
        sum = sum + item;
   }

   printf("Consumer: SUM=%d\n", sum);
   pthread_exit((void *)0);
}

void enter_item(int item)
{
/* Wait the condition full: buffer is  full, count is equal to BUFS */
      pthread_mutex_lock(&mut);
      while( count == BUFS )
             pthread_cond_wait(&full, &mut);

      buffer[wp] = item;
      wp = (wp + 1)%BUFS;

/* Signal the condition empty: buffer is not empty, count is 1 */
      count = count + 1;
```

```
      if (count == 1) pthread_cond_signal(&empty);
      pthread_mutex_unlock(&mut);

     return;
}

int remove_item(void)
{
  int item;

/* Wait the condition empty: buffer is empty, count is 0 */
      pthread_mutex_lock(&mut);
      while( count == 0 )
          pthread_cond_wait(&empty, &mut);
      item = buffer[rp];
      rp = (rp + 1)%BUFS;

/* Signal the condition full:buffer is not full, count is equal to BUFS-1*/
      count = count - 1;
      if (count == BUFS-1) pthread_cond_signal(&full);
      pthread_mutex_unlock(&mut);
      return(item);
}
/* ------------------------------------------------------------ */
```

Производителят изпраща на Потребителя последователни цели числа, който ги сумира и накрая извежда резултата на стандартния изход. Функциите `th_prod` и `th_cons`, изпълнявани в двете нишки, получават като аргумент броя на елементите, които ще си предават. Изпълнихме програмата няколко пъти и получихме следния изход.

```
$ a.out 100
Consumer: SUM=5050
$ a.out 1000
Consumer: SUM=500500
$ a.out 10000
Consumer: SUM=50005000
```

Реализацията на POSIX нишките в различните версии на Unix и Linux се различава. В някои версии нишките са реализирани като процеси, т.е. когато извикваме `pthread_create` за да създадем нова нишка, се създава нов процес с нов pid, който да изпълнява нишката. Но този процес се различава от процес създаден чрез `fork`. Процесите за двете нишки (оригиналната и новосъздадената) имат общо адресно пространство и други ресурси. В по-новите версии на Linux нишките в един процес имат еднакъв pid.

Пример

Програма 8.5 ще ни демонстрира каква е реализацията на нишките. Главната нишка създава друга нишка, двете нишки извикват `getpid` и извеждат на стандартния изход значението на pid си, след което изпълняват безкраен цикъл.

```
/* ------------------------------------------------------------ */
/*  Threads - Test for pid of thread */

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

main(void)
{
  pthread_t th1;
  void *print_pid();
```

```
  printf("Main pid=%d\n", getpid());
  pthread_create(&th1, NULL, print_pid, NULL);
  while(1);

  pthread_join(th1, NULL);
  exit(0);
}

/* Thread function */
void *print_pid()
{
  printf("Thread pid=%d\n", getpid());
  while(1);

  pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Изпълняваме програмата във фонов режим и чрез командата `ps` проверяваме процесите. В Linux Kernel 2.0 получихме следния резултат.

```
$ a.out &
Main pid=481
Thread pid=483
$ ps
  PID TTY STAT TIME COMMAND
  424   2 S    0:00 -bash
  481   2 R    0:22 a.out
  482   2 S    0:00 a.out
  483   2 R    0:22 a.out
  484   2 R    0:00 ps
```

Виждаме, че има три процеса за програмата, въпреки че нишките са две. Първият процес, с pid 481, е за главната нишка. Третият процес, с pid 483, е за създадената с `pthread_create` нишка. Процесът с pid 482 реализира допълнителна нишка, наречена “manager thread”. Той е част от реализацията на нишките в Linux, създава се при първото извикване на `pthread_create` и е баща на така създадените нишки. Следователно, главната нишки и всички останали в процеса имат различен ppid. (Не трябва да забравим да прекратим процесите с команда `kill`.)

В Linux Kernel 2.6 получихме друг резултат.

```
$ a.out &
Main pid=30680
Thread pid=30680
$ ps
  PID TTY          TIME CMD
29128 pts/1    00:00:01 -bash
30680 pts/1    00:01:00 a.out
30690 pts/1    00:00:00 ps
```

Виждаме, че двете нишки имат еднакъв pid 30680 и командата `ps` показва един процес за програмата. Освен това всички нишки в процеса имат еднакъв ppid.

Приложение 1

ФУНКЦИИ НА СИСТЕМНИТЕ ПРИМИТИВИ

Файлова система

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

int creat(const char *filename, mode_t mode);
int open(const char *filename, int oflag [, mode_t mode]);
int close(int fd);
ssize_t read(int fd, void *buffer, size_t nbytes);
ssize_t write(int fd, void * buffer, size_t nbytes);
off_t lseek(int fd, off_t offset, int flag);
int dup(int fd);
int dup2(int fd, int newfd);
int fcntl(int fd, int cmd, long arg);

#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);

mode_t umask(mode_t cmask);
int chmod(const char *filename, mode_t mode);
int fchmod(int fd, mode_t mode);
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);
int utime(const char *filename, const struct utimbuf *times);

int link(const char *oldname, const char *newname);
int symlink(const char *toname, const char *fromname);
int readlink(const char *filename,char *buf,size_t bufsize);
int unlink(const char *filename);
int mkdir(const char *dirname, mode_t mode);
int rmdir(const char *dirname);
int chdir(const char *dirname);
int fchdir(int fd);
int mknod(const char *name , mode_t mode, dev_t dev);
int mount(const char *special, const char *dirname,
          const char *fstype,unsigned long flags,void *data);
int umount(const char *dirname);
int fsync(int fd);
void sync(void);
```

Библиотечни функции за четене от каталог

```
#include <dirent.h>

DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
void rewinddir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
```

Процеси

```
pid_t fork(void);
void _exit(int status);
void exit(int status);
int atexit(void (*function)(void));

#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

     WIFEXITED(status), WEXITSTATUS(status)
     WIFSIGNALED(status), WTERMSIG(status), WCOREDUMP(status)
     WIFSTOPPED(status), WSTOPSIG(status)

int execl(const char *name, const char *arg0
          [,const char *arg1]..., 0);
int execlp(const char *name, const char *arg0
          [,const char *arg1]..., 0);
int execle(const char *name, const char *arg0
          [,const char *arg1]..., 0, char *envp[]);
int execv(const char *name, char *argv[]);
int execvp(const char *name, char *argv[]);
int execve(const char *name, char *argv[], char *envp[]);
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
int setuid(uid_t uid);
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
int setpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
pid_t getsid(pid_t pid);
pid_t setsid(void);
```

Сигнали

```
#include <signal.h>

void (*signal(int sig, void (*sighandler)(int)))(int);
int kill(pid_t pid, int sig);
int pause(void);
unsigned int alarm(unsigned int sec);
```

Програмни канали

```
int pipe(int fd[2]);
```

Променливи на обкръжението

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *str);
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```

Съобщения

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflag);
int msgsnd(int msgid, struct msgbuf *msg, size_t size, int msgflag);
int msgrcv(int msgid, struct msgbuf *msg, size_t size,
           long type, int msgflag);
int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

Обща памет

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflag);
void *shmat(int shmid, void *addr, int shmflag);
int shmdt(void *addr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Семафори

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflag);
int semop(int semid, struct sembuf *sops, unsigned nsops);
int semctl(int semid, int semnum, int cmd, union semun arg);

struct sembuf {
   ushort sem_num;
   short  sem_op;
   short  sem_flg; };
```

Нишки

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **value_ptr);
int pthread_detach(pthread_t thread);

int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *condattr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cancel(pthread_t thread);
void pthread_testcancel(void);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

Приложение 2

ЗАГЛАВЕН ФАЙЛ `ourhdr.h`

В повечето примери е използван заглавния файл `ourhdr.h`. Той включва системни заглавни файлове, които са нужни на болшинството примери.

Тук са и определенията на нашите функциите за обработка на грешки. Функциите приемат произволен брой аргументи и извеждат съобщение на стандартния изход за грешки. Разликите между различните функции са обобщени в следната таблица.

Функция	Използа `errno`	Процесът завършва
`err_sys_ret`	Да	Не
`err_sys_exit`	Да	Да
`err_ret`	Не	Не
`err_exit`	Не	Да

Използването на тези функции съкращава текста на програмите при обработка на грешките. Проверката изглежда така (като вместо `err_sys_exit` може да друга от функциите).

```
if (има грешка)
   err_sys_exit(printf формат с произволен брой аргументи);
```

Съдържание на заглавен файл `ourhdr.h`.

```
/*-------------------------------------------------------------*/
#include <unistd.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#define MAXLINE 1024

void err_doit(int, const char *, va_list);

/* After error related to system call.
 * Print a message and exit */

void err_sys_exit(char * frm, ...)
{
  va_list ap;
  va_start(ap, frm);
  err_doit(1, frm, ap);
  va_end(ap);
  exit(1);
}

/* After error related to system call.
 * Print a message and return */

void err_sys_ret(char * frm, ...)
{
  va_list ap;
  va_start(ap, frm);
```

```
  err_doit(1, frm, ap);
  va_end(ap);
  return;
}

/* After error unrelated to system call.
 * Print a message and exit */

void err_exit(char * frm, ...)
{
  va_list ap;
  va_start(ap, frm);
  err_doit(0, frm, ap);
  va_end(ap);
  exit(1);
}

/* After error unrelated to system call.
 * Print a message and return */

void err_ret(char * frm, ...)
{
  va_list ap;
  va_start(ap, frm);
  err_doit(0, frm, ap);
  va_end(ap);
  return;
}

/* Print a message and return to caller */

void err_doit(int errflag, const char *frm, va_list ap)
{
  int err_save;
  char msg[MAXLINE];
  err_save = errno;
  vsprintf(msg, frm, ap);
  if (errflag)
    sprintf(msg+strlen(msg), ": %s", strerror(err_save));
  strcat(msg, "\n");
  fflush(stdout);
  fputs(msg, stderr);
  fflush(NULL);
  return;
}
/*------------------------------------------------------------*/
```

ЛИТЕРАТУРА

1. Bach M.J. The Design of UNIX Operating System. Prentoce Hall, Englewood Cliffs, N.J., 1986.
2. Stevens W.R. Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, Mass, 1992.
3. Stevens W.R. UNIX Network Programming, Volume 2, Second Edition, Inteprocess Communications, Prentice Hall, Upper Saddle River, N.J., 1999.
4. Стивенс У. UNIX: взаимодействие процессов. СПб.: Питер, 2003.