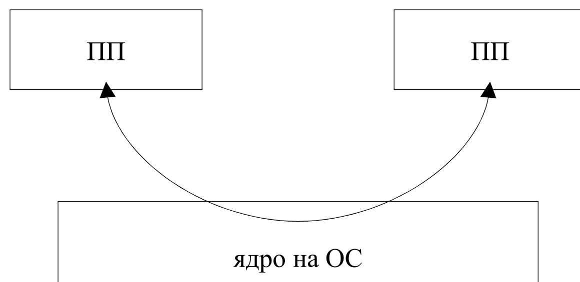
7. КОМУНИКАЦИИ МЕЖДУ ПРОЦЕСИ

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) на UNIX System V
- обща памет (shared memory) на UNIX System V
- семафори (semaphores) на UNIX System V

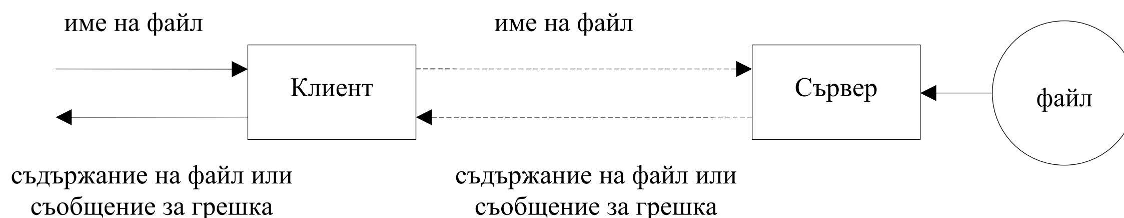
Когато два или повече процеса използват някакъв механизъм за обмен на информация, то IPC обекта трябва да има име или идентификатор. Така един от процесите ще създаде IPC обекта, а останалите ще получат достъп към този конкретен обект. Множеството от имена за определен вид IPC, наричаме пространство 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

Фиг. 7.2. Пространство на имената за видовете IPC

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, отваря файла за четене. При успех сърверът 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.



Фиг. 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. Виж ограничението в `PIPE_BUF` от Програма 1.3 (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`.

```
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 от капацитета на канала (от `PIPE_BUF`), 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 размера на файла с прочетения брой байта и събужда 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 `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 EOF).

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.

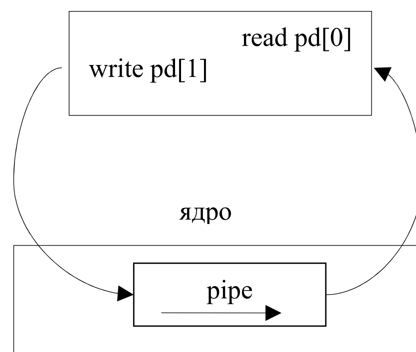
```
/* ---------------------------------------------------------- */
/* Simple example of pipe in one process */

#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 напълни 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.

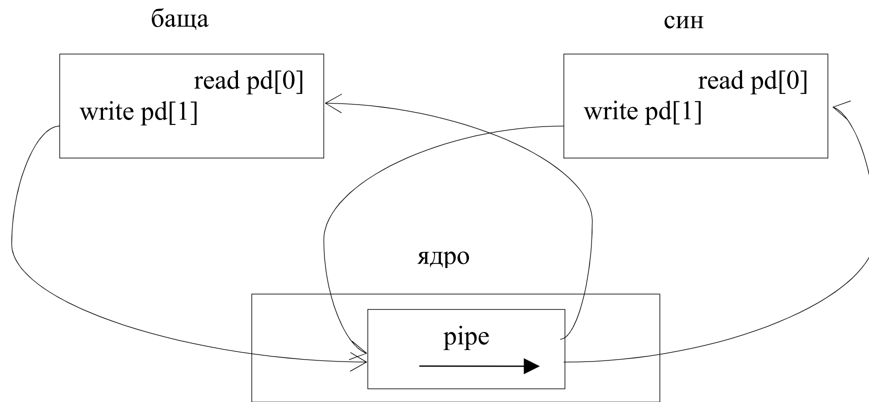


Фиг. 7.5а. Програмен канал в един процес веднага след `pipe`

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

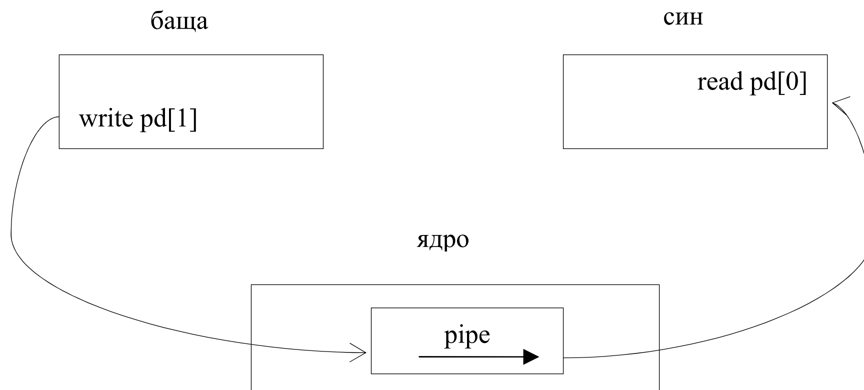
- процесът създава програмен канал - изпълнява `pipe`
- процесът създава нов процес - изпълнява `fork`

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 Фиг.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 пуска програмата `wc`, който 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 първия си процес-син (`wc`).

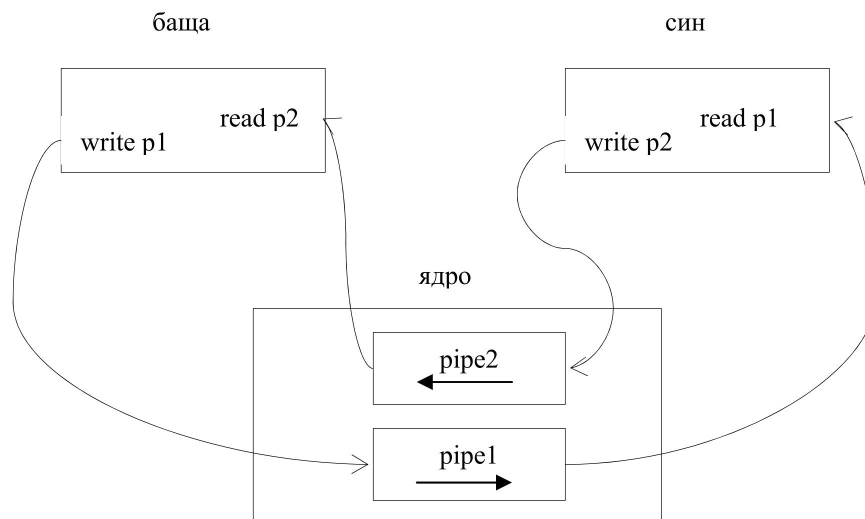
```
/* ------------------------------------------------------------ */
/* Pipe Line ls | wc -l */

#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 сърверът 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");

  buff[n] = '\0';
  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 съответния примитив `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 (полето `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 */
      ushort uid;         /* owner's user ID */
      ushort gid;         /* owner's group ID */
      ushort cuid;        /* creator's user ID */
      ushort cgid;        /* creator's group ID */
      ushort mode;        /* access mode */
      ushort 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 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 и поле mode

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 полето 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. (Например, 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 Scientific Linux 2.6.9-22.0.2.EL 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`), също 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 `<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 типът 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 няма съобщение в опашката 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 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 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 Red Hat Linux 4.1 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 Scientific Linux 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);
     }
  }
}

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.

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 сърверът 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 7.8а, 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("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. Следва изхода 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
```

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 Scientific Linux 2.6.9-22.0.2.EL 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 SMS) 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 */
      int 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 SMS: права на достъп и собственост (`shm_perm`), размер на общата памет (`shm_segsz`), времена на последни операции над SMS (`shm_atime`, `shm_dtime`, `shm_ctime`), брой процеси, присъединили SMS към адресното си пространство (`shm_nattch`) и идентификатори на процеси, изпълнили операции над SMS (`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 SMS 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 SMS, 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 SMS.

След успешно изпълнение на `shmget`, процесът първо трябва да присъедини SMS 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, SMS 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 SMS в процеса при успех, -1 при грешка.

```
int shmdt(void *addr);
```
Връща 0 при успех, -1 при грешка.

Функцията `shmat` присъединява SMS 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 SMS. При успех функцията връща действителния виртуален адрес на SMS в процеса.

Функцията `shmdt` освобождава SMS, 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 SMS 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 SMS, промяна на някои атрибути, като правата на достъп и собственик и унищожаване на SMS.

```
#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 SMS. След `exec` всички присъединени SMS се освобождават (не унищожават). При завършване на процес (`exit`) присъединените SMS 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 ((int)shm_adr == -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 Red Hat Linux 4.1 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 Scientific Linux 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 SMS */

#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, 0);
  if ((int)shm_adr == -1)
     err_sys_exit("shmat error");

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

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

/* Destroy SMS */
  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

Програма 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 ((int)ptr == -1)
     err_sys_exit("race1: shmat failed");

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

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

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, 0);
  if ( (int)ptr == -1 )
     err_sys_exit("race1: 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 race , след като  race2 завърши>
$ Race1 quits
[1]+  Done                   race1
```

Проблемът при използването на обща памет за комуникация между процеси (демонстриран в Програма 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 Дейкста 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 Scientific Linux 2.6.9-22.0.2.EL 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 */
  ushort          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 семафора (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 Scientific Linux нищо не се говори за началната инициализация при създаването.

След инициализацията на семафора, над него се изпълняват операции чрез 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 {
   ushort  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 споменем 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.

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.

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 Scientific Linux 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 = Mon Sep 10 17:11:58 2007
semnum     value      ncount     zcount     pid
0          1          0          0          30322

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

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;
};
static 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, 0L, 0);
  write(fd, &i, sizeof(int));
}

/* Read from shared file */
int get_shared(void)
{
  int i;
  lseek(fd,0L,0);
  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[0], 2) == -1)
    err_sys_exit( "P(%d) failed", semid);
}

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

  if (semop(semid, &vsem[0], 1) == -1)
     err_sys_exit("V(%d) failed", semid);
}
/* ------------------------------------------------------------ */
```