12. Управление на процеси и междупроцесни комуникации.

Основни системни примитиви за управление на процеси. Създаване на процес. Изпълнение на програма. Завършване на процес. Синхронизация със завършването на процеса -син. Права на процеси – потребителски идентификатори на процес. Групи процеси и сесия. Механизми за междупроцесни комуникации. Сигнали. Програмни канали. IPC пакет на UNIX System V: Обща памет. Семафори. Съобщения.

УПРАВЛЕНИЕ НА ПРОЦЕСИ

➢ ОСНОВНИ СИСТЕМНИ ПРИМИТИВИ ЗА УПРАВЛЕНИЕ НА ПРОЦЕС

Процес е програма в хода на нейното изпълнение**.** Програма на 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.

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):

1 - идентификатор на процеса (**pid**)
2 - идентификатор на процеса-баща (**ppid** от parent pid)
3 - идентификатор на група процеси
4 - идентификатор на сесия
5 - реален потребителски идентификатор (**ruid**)
6 - ефективен потребителски идентификатор (**euid**)
7 - реален идентификатор на потребителска група (**rgid**)
8 - ефективен идентификатор на потребителска група (**egid**)
9 - файлови дескриптори на отворените файлове
10 - текущ каталог
11 - управляващ терминал
12 - маска, използвана при създаване на файлове и заредена с примитива umask
13 - реакция на процеса при получаване на различни сигнали
14 - времена за изпълнение на процеса в системна и потребителска фаза (system and user CPU time)

Съществуват системни примитиви, които връщат значенията на различните атрибути на процеса, който ги извиква. Следват прототипите на функциите за част от тях. 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.
uid_t getuid(void);          Връща ruid на процеса.
uid_t geteuid(void);         Връща euid на процеса.
gid_t getgid(void);          Връща rgid на процеса.
gidt getegid(void);          Връща egid на процеса.
```

➢ СЪЗДАВАНЕ НА ПРОЦЕС

- Системен примитив **`fork`**

Единственият начин за създаването 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`.

Заб: Не е задължително

```
/* Example of fork */
#include <sys/types.h>
#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 `sleep`, 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. Този начин за синхронизация между два процеса не е най-добрият, по-нататък ще разгледаме други механизми. наследява от бащата файловите дескриптори.

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

1 - идентификатор на група процеси
2 - идентификатор на сесия
3 - реален потребителски идентификатор
4 - ефективен потребителски идентификатор
5 - реален идентификатор на потребителска група
6 - ефективен идентификатор на потребителска група
7 - файлови дескриптори на отворените файлове
8 - текущ каталог
9 - управляващ терминал
10 - маска, използвана при създаване на файлове
11 - реакция на процеса при получаване на различни сигнали

12 - променливи от обкръжението

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

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

Има два 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 сървъри.

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 при командните интерпретатори.

➢ 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 (kill)
- 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 файла 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 (*fucntion)(void));
```

Връща 0 при успех, -1 при грешка.

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 кодът на завършване, 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`.

➢ СИНХРОНИЗАЦИЯ СЪС ЗАВЪРШВАНЕТО НА ПРОЦЕСА-СИН – **wait** и **waitpid**

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:
   - 0 - *pid* == -1 Чака първия син, който завърши.
   - 1 - *pid* > 0 Чака син с идентификатор *pid*.
   - 2 - *pid* == 0 Чака първия син от същата група процеси.
   - 3 - *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 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:

1 - Как е завършил сина - нормално или аварийно?

2 - Какъв е кода на завършване, предаден в `exit`, или номера на сигнала, който е предизвикал аварийното му завършване?

3 - Създаден ли е файл с име `core` (някои сигнали предизвикват създаването му)? (core dump file when the program crashes – позволява debug на програмата, която е crash-нала)

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).

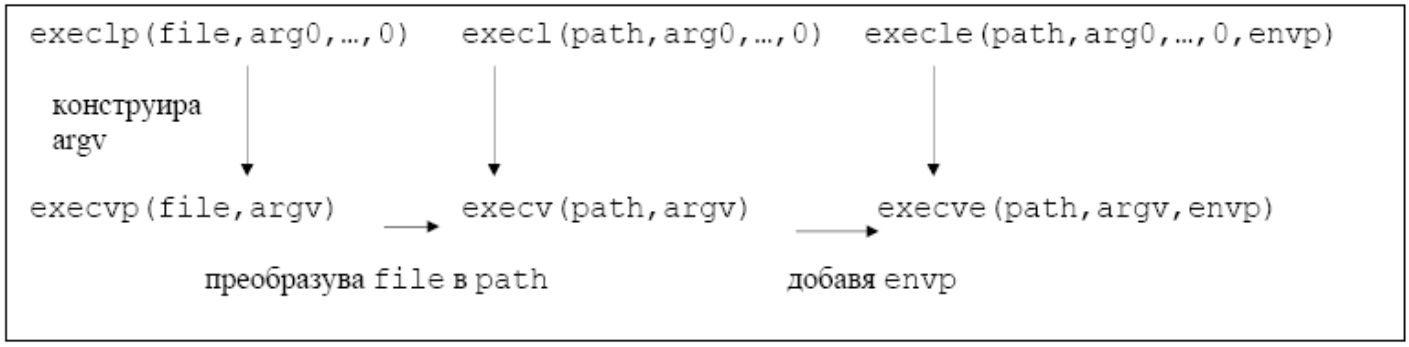
➢ ИЗПЪЛНЕНИЕ НА ПРОГРАМА - **exec**

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

Връща -1 при грешка, нищопри успех

Връзката между функциите на примитива `exec` е следната:



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 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 обкръжението на процеса и се използва значението на глобалната променливата environ.

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 елементите на контекста му. При грешка по време на exec става връщане в става връщане в стария образ, така че функцията връща -1 при грешка, а при успех не връща нищо, защото няма връщане в стария образ.

➢ ПРАВА НА ПРОЦЕСИ – ПОТРЕБИТЕЛСКИ ИДЕНТИФИКАТОРИ НА ПРОЦЕС

С всеки процес са свързани два потребителски идентификатора: реален - ruid и ефективен - euid. Реалният е идентификаторът 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. Всъщност се пази още един ефективен идентификатор (в таблицата на процесите), 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.

Промяна на ефективния идентификатор може да стане по два начина:

- Когато се изпълнява системен примитив `exec` и програмата е set-UID (set-UID - “change uid when run”) 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 `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.
- 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 параметъра *uid*. Ако текущият euid на процеса не е 0, то се променя euid с параметъра *uid*, само ако параметърът *uid*, е равен на ruid или на suid.

➢ ВРЪЗКИ МЕЖДУ ПРОЦЕСИ – ГРУПИ ПРОЦЕСИ и СЕСИЯ

• ГРУПИ ПРОЦЕСИ

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.

```
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:

```
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. Примитивът `setpgid(0,0)` има същото действие както `setpgrp()`. При успех и двете функции връщат 0.

• СЕСИЯ

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.

```
pid_t getsid(pid_t pid);
```
Връща идентификатор на сесия при успех, -1 при грешка.

```
pid_t setsid(void);
```
Връща 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. 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 U area. Какво съдържа това поле? Всеки терминал има свързана с него tty структура. Полето за управляващ терминал в U area на процеса е указател към tty структурата на управляващия му терминал. Ако това поле има значение `NULL`, то процесът няма управляващ терминал.

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. Това поле определя групата процеси, на които се изпращат сигнали, свързани с терминала: `SIGINT`, `SIGQUIT`, `SIGHUP`, `SIGTSTP`, `SIGCONT` (първите три са общи за всички Unix и Linux системи, последните два са от 4.3BSD). Така, когато въведем <Ctrl>+<C> на терминала, ядрото изпраща сигнал `SIGINT` на всички процеси от привилегированата група. Освен това, входа 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.

Междупроцесорни Комуникации

➢ СИГНАЛИ

Сигналите формират 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 междупроцесорна комуникация, но също така много напомнят механизма на прекъсвания.

- Типове сигнали

Всеки сигнал има уникален номер и символно име, които определят събитието, за което информира сигнала. Типовете сигнали могат да се класифицират в зависимост от събитието, свързано със сигнала:

1. Сигнали, свързани с управляващия терминал – Изпращат се на процеса(процесите от привилегированата група), свързан с терминала.
   - SIGINT – изпраща се, когато потребителят натисне клавиш <Del> или <Ctrl>+<C>(Signal Interruption)
   - SIGQUIT – изпраща се, когато потребителят натисне клавиш <Ctrl>+</> (Quit)
   - SIGHUP – изпраща се при прекъсване на връзката с управляващия терминал (Hang up)
2. Сигнали, свързани с апаратни особени ситуации – Свързани са със събития, откривани от апаратурата и сигнализирани чрез прекъсване.
   - SIGFRE – изпраща се при делене на нула или препълване на операции с плаваща точка (floating point exception)
   - SIGILL – изпраща се при опит за изпълнение на недопустима инструкция (illegal operation)
   - SIGSEGV – изпраща се при обръщение към недопустим адрес или към адрес, за който процесът няма права (segmentation violation)

3. Сигнали, свързани с програмни ситуации- Имат чисто програмен характер и не се сигнализират от апаратурата.

SIGCHLD – изпраща се на процес-баща, когато някой негов процес-син завърши (child)
SIGALRM – изпраща се, когато изтече времето, заредено за процеса, чрез системния примитив alarm
SIGPIPE – изпраща се при опти на процес да пише в програмния канал, който вече не е отворен за четене

Процес може да изпрати на друг процес сигнал чрез системния примитив kill

SIGKILL – предизвиква безусловно завършване на процеса (kill)
SIGTERM – предупреждение за завършване на процеса (terminate)
SIGUSR1 – потребителски сигнал, използван от потребителските процеси като средство за междупроцесорни комуникации (user-defined signal 1)
SIGUSR2 – още един потребителски сигнал(user-defined signal 2)

- Изпращане и обработка на сигнали

Сигнал може да бъде изпратен на процес или от ядрото или от процес чрез системния примитив kill. Ядрото помни изпратените, но още необработениот процеса сигнали в записа от таблиците на процесите. Полето е масив от битове, в който всеки бит отговаря на тип сигнал. При изпращане на сигнал съответния бит се вдига.
Има три възможни начина за обработка на един сигнал, ще ги наричаме реакции на сигнал:

- Процесът завършва (реакция 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 сигнала

Реакцията за всички видове сигнали се помнив потребителската област на процеса, където полето е масивот адресите на обработчиците на сигналите, един за всеки тип сигнал.

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

Определя реакцията на процес при получаване на сигнал от определен тип (различна от тази по премълчаване). Аргументът sig задава номера на сигнала, а sighandler определя каква да е реакцията при получаване на сигнал:

- SIG_IGN – игнориране на сигнал - signal_ignore
- SIG_DFL – реакция по премълчаване(обикновено завършване на процеса) (default_signal)
-име на потребителската функция

Реакцията се запомня в съответното поле от потребителската област и с това работата на примитива приключва.
Процес-син наследява реакциите на сигнали от процеса-баща. След exec за всички сигнали, за които реакцията е била променяна с потрбителска фунцкия, се връща реакция по премълчаване. Това е естественото поведение, тъй като при exec се сменя образа на процеса.

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

Чрез примитива kill се извършва изпращането на сигнал на един процес към друг(и). Аргументът sig задава на номера на типа на сигнал, който се изпраща. Аргументът pid определя процесите, на които се изпраща сигнала. Възможни 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 от таблицата на процесите (без процеса init (pid=1) и системните процеси)

```
int pause(void);
```
Връща -1.

Системният примитив pause блокира процеса, който го изпълнява до получаване на първия сигнал, за който реакцията не е игнориране. Ако реакцията за първият пристигнал сигнал е SIG_DFL, то процесът завършва, т.е. връщане от pause няма. Ако за сигнала е предвидена потребителска функция и от тази фунцкия има връщане, то след изпълнение на функцията процесът продължава от оператора след pause. В този случай има връщане от pause и функцията връща -1.

```
unsigned int alarm(unsigned int sec);
```
Връща 0 или число по-голямо от 0.

Системният примитив alarm планира изпращането на сигнал SIGALRM на процеса, изпълняващ примитива. Аргументът sec задава брой секунди, т.е. при изпълнението на alarm в таймера на процеса се зарежда значението му. 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 броят секунди, оставащи до него. В противен случай функцията връща 0. Чрез изпълние на примитива с аргумент sec, равен на 0, може да се анулира зареден преди това таймер без да се планира нов сигнал.

- Комуникации между процеси

IPC е съкращение от Interprocess Communication или комуникация междъ процеси. Когато два или повече кокурентни 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. С развитието на операционните системи от тип UNIX и Linux в тях са включени методи и механизми за междупроцесорна комуникация:

- програмни канали (pipes)
- именовани програмни канали (named pipes/ FIFO files)
- 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 IPC, наричаме пространство от имена. Примерно pipes нямат имена, а FIFO files използват име на файл, след отваряне и двата вида IPC използват идентификатор – файлов дескриптор.

- Програмни канали

Програмният канал е механизъм за комуникация между процеси, който осигурява еднопосочно предаване на неформатиран поток от данни (поток от байтове) между процеси и синхронизация на работата им. Реализат се два типа канала в UNIX и Linux:

- наименован програмен канал (pipe) – за коминкация между родствени процеси
- именован програмен канал (named pipe или FIFO файл) – за комуникация между независими процеси

Като механизъм на комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове:

- за четене и писане в него се използват системните примитиви read/write,но дисциплината е FIFO
- каналът има доста ограничен капацитет, различен в различните реализации

Операции над 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

```
int pipe(int fd[2]);
```
Връща 0 при успех, -1 при грешка.

Програмен канал се създава чрез примитива pipe. Създава се нов файл от тип програмен канал, който включва разпределяне и инициализиране на свободен индексен описате, както и при обикновените файлове, но за разлика от тях, каналът няма външно име и 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 файлове дескриптор за писане в fd[1].
Писане и четене в програмен канал се извършва с примитивите writeи read, но достъпът до данните е FIFO.

write - Писане в програмен канал

1. Ако каналът има достатъчно място , то данните се записват в края на файла, увеличава се размера на файла със записания брой байтове и се събуждат всички процеси, чакащи да четат канала.
2. Ако няма достъчно място и броят байтове за запис е по-малък от капацитетът на канала, ядрото блокира процеса. Събужда се от друг процес с read и продължава като в случай 1.
3. Ако няма достъчно място и броят байтове за запис е по-голям от капацитетът на канала, то в каналът се записват толкова байта, колкото е възможно и ядрото блокира процеса. Когато бъде събуден, процесът 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.

read - Четене в програмен канал

1. Ако в канала има някакви данни, то започва четене от началото на файла, докато се изпълни целта или се стигне до края на файла, намалява се размера на файла и се събуждат всички процеси, чакащи да пишат в канала.
2. Ако каналът е празен, ядрото блокира процеса. Когато бъде събуден от друг процес с write, продължава като в случай 1.

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

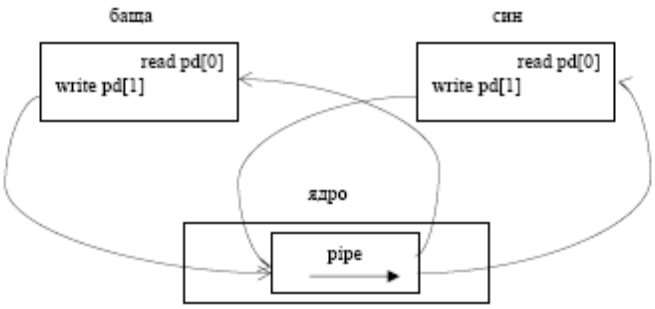
close - Затваряне на програмен канал

1. Ако при close се освободи последния файлов дескриптор за писане в канала (на всички процеси), то се събуждат всички процеси, чакащи да четат в канала, като read връща 0 (eof).
2. Ако при close се освободи последния файлов дескриптор за четене от канала, то се събуждат всички процеси, чакащи да пишат в канала като им се изпраща сигнал SIGPIPE.
3. При освобождаване и на последния fd за работа с канала, програмния канал се унищожава.

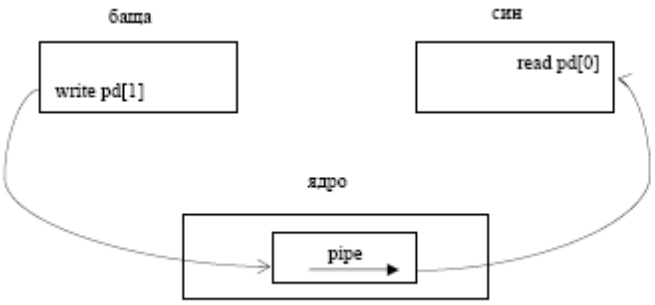
Буфериране!!!

printf – fully buffered, когато stdout е пренасочен към файл
- line buffered, когато stdout е на терминала (до '\n')

write - не се буферира



Програмен канал между два процеса веднага след `fork`



Еднопосочна комуникация между два процеса с програмен канал

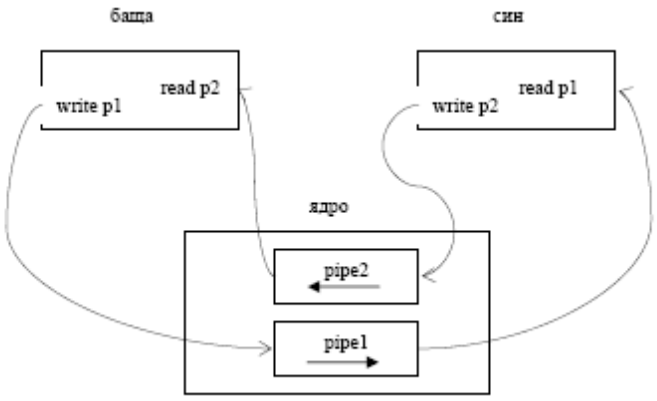
Еднопосочна комуникация между два процеса

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

В резултат има два процеса – баща и син, като синът е наследил от баща си двата файлови дескриптора за програмния канал. Ако след това бащата затвори файловия дескриптор за четене, а синът затвори този за писане (може и обратното), ще се получи еднопосочен канал между двата процеса – от бащата към сина.

Двупосочна комуникация между два процеса



Двупосочна комуникация между два процеса с програмни канал

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

В резултат има два процеса – баща и син, като синът е наследил от баща си двата файлови дескриптора

➢ IPC пакет на UNIX System V

IPC пакетът на UNIX System V включва три механизма: съобщения, обща памет и семафори. Има общи черти при реализацията на тези три 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

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 не IPC обекта (аналог на името на файл). При създаването на IPC обекта ядрото преобразува ключът във вътрешен **идентификатор** (аналог на файловия дескриптор при файлове). Този идентификатор се използва от примитивите за работа с IPC.

Всеки IPC обект се представя в ядрото чрез структура, съдържаща информация за него. Независимо от вида на IPC обектатази 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 */
};
```

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 (аналог на кода за защита, но се използват само битовете r и w). Чрез функциите XXXctl могат да бъдат изменени uid, gid и mode. Така елементите uid и gid съдържат идентификатори на текущия собственик на обекта. Елементите cuid и cgid не могат да бъдат изменяни, т.е. те винаги съдържат идентификатори на създателя на обекта. Функциите XXXctl са аналог на chown и chmod при файлове.

➢ Съобщения (Message Passing)

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

- **Косвена адресация** – съобщенията се предават в и получават от опашка на съобщенията (message queue)
- **Автоматично буфериране** – съществува системен буфер с ограничен капацитет, в който временно могат да се съхраняват изпратени и още неполучени съобщения

Всяка опашка на съобщения се представя в ядрото чрез структура msqid_ds, определена в заглавния файл

```
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 */
};
```

<sys/msq.h> и съдържа следните елементи.

Елементите на структурата съдържат информация за опашката: права на достъп и собственост (msg_perm-permissions ;]), брой съобщения в опашката (msg_qnum) и обща дължина на всички съобщения в нея (msg_cbytes), време и идентификатор, изпълнил последния msgsnd (msg_stime, msg_lspid) и същото за msgrcv (msg_rtime, msg_lspid). Елементът msg_qbytes определя максималната обща дължина на всички съобщения в опашката.

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

int msgget(key_t key, int flag);
                        Връща идентификатор на MQ при успех, -1 при грешка.
```

Опашката на съобщения се създава или отваря чрез функцията msgget. Аргументът key съдържа ключа (външното име) за опашката. Нов обект MQ се създава, ако е изпълнено едно от условията:

- не съществува MQ с ключа key и в аргументът flag е зададено IPC_create;
- в аргумента flag 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 й.

```
#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 събощение се състои от тип-цяло положително числои 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 */
};
```

Обикновено програмата сама трябва да определи своя структура за съобщението, в която елементът mtext е с нужната дължина (а не 1 винаги).

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 0). Последният аргумент flag определя каква да е реакцията, ако съобщението не може да бъде изпратено веднага: има много съобщения в опашката или въобще в системата.

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 съобщенията в MQ ще бъде получено:

- type = 0 – първото съобщение в MQ
- type > 0 – първото съобщение от тип type в MQ
- type < 0 – първото съобщение от тип най-малкото число <= |type|

По този начин чрез типа на съобщението и описаното действие на msgrcv, могат да се реализират няколко потока

```
#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.
```

за предаване на съобщения в рамките на една опашка на съобщения, включително и двупосочна коминкация.

Функцията msgctl реализира операциите по управление на опашката от съобщения: получаване на информация за състоянието на MQ, промяна на някои атрибути като права наа достъп и собственик, унищожаване на MQ.

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

- IPC_RMID – унищожава се опашката с идентификатор msgid
- IPC_STAT – получава се информация за опашката с идентификатор msgid чрез аргумента buf
- IPC_SET – изменят се атрибути на опашката с идентификато msgid (като msg_perm.uid/mdg_perm.mode, msg_qbyte и т.н.) като аргументът buf определя новите значения

IPC обект съществува, докато не се унищожи явно чрез съответната функция XXXctl. Унищожаването на опашка от съобщениясе извършва незабавно, т.е. при изпълнение на функция msgctl. Ако има процеси, които са блокирани в msgrcv или msgsnd, те се събуждат и съответната фунцкия връща 01 и код EIDRM в errno. Така е и при унищожаване на семафори, но не и при обща памет.
NB!!!Чрез ipcs -q се получава информация за message queues.

➢ Обща памет

Общата памет е най-бързият и прост метод за комуникация 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 данните, 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.
Всеки сегмент обща памет (IPC обект = 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 обща памет (shm_segsz), времена на последните операции над SMS (shm_atime, shm_dtime, shm_ctime), брой процеси, присъединили SMS към адресното си пространство (shm_nattach) и идентификатори на процеси, изпълнили операции над SMS (shm_cpid, shm_lpid).

```
#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.
```

Сегментът обща памет се създава или отваря чрез функцията shmget. Аргументите key и flag имат същото значение и логика както при останалите функции XXXget. Аргументът size задава размера на създавания сегмент обща памет. При успех функцията връща вътрешен идентификатор, който се използва от останалите функции за идентифициране на SMS.

```
#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.
```

След успешно изпълнение на shmget, процесът първо трябва да присъедини SMS към адресното си пространство чрез функцията shmat (attach). След това може да чете и пише в общата памет. Когато не е необходим, SMS се освобождава чрез функцията shmdt (detach).
Функцията shmat присъединява SMS с идентификатор shmid към адресното пространство на процеса. Адресът за присвояване се определя от аргументире addr и flag.

addr = 0 – ядрото определя адреса (препоръчителен начин)
addr != 0 – задава се виртуален адрес за присвояване
flag = SHM_RND – адресът се подравнява на граница, зададена от ядрото
flag = SHM_RDONLY – сегментът се присъединява само за четене, иначе и за четене и писане

Функцията shmdt освобождава SMS, присъединен преди това от процеса на адрес addr от адресното пространство на процеса. Като аргумент на функцията се задава адреса на сегмента обща памет, а не идентификатор shmid. Причината за това е, че един SMS може да бъде присъединен няколко пъти към адресното пространство на един

```
#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.
```

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 права на достъп и собственик и унищожаване на SMS.
Операциите в аргумента cmd са същите както при съобщенията: IPC_STAT, IPC_SET, IPC_RMID. При операцията 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 структурата shmid_ds стане 0).
Унищожаване на SMS – различно от messages & semaphores
След fork процес-син наследява от бащата всички присъединени SMS. След exec всички присъединени SMS се освобождават (не унищожават). При завършване на процеса (exit) присъединените SMS автоматично се освобождават.

➢ Семафори (Semaphores)

Семафорите са предложени от Дийкстра като механизъм за осигуряване на взаимно изключване и синхронизация на процеси, използващи общи ресурси.
Особености на семафорите в IPC пакета на UNIX System V:

- 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 (семафор) има ключ и идентификатор и се представя в ядрото чрез структурата 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

```
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 семафор
semncnt – брой процеси, чакащи да се увеличи значението на семафора
semnzcnt – брой процеси, чакащи значението на семафора да стане 0

```
#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.

Масив от семафори се създава или отваря чрез функцията shmget. Аргументите key и flag имат същото предназначение, както при другите два IPC механизма. Аргументът nsems задава броя на елементите в масива семафори. В някои реализации на UNIX и Linux semval може да се инициализира с 0, но в други това не се прави и трябва се направи чрез функцията semctl с команда SETVAL или SETALL. Така инициализацията на семафор вече не е атомарна операция – основен недостатък.

```
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```
Връща 0 при успех, -1 при грешка.

Чрез функцията semop се изпълняват операции над семафорите. 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. SEM_UNDO – ядрото осигурява отмяна на операцията при завършване на процеса; IPC_NOWAIT + 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 операции.

```
#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.

Функцията semctl реализира операциите по управление на семафори: получаване на информация за състоянието на семафори; инициализация на брояча; промяна на атрибути като права на достъп и собственик; унищожаване на масива семафори.
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 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 */
};
```

Основни операции в аргумента cmd са следните:

- IPC_RMID – унищожава се масив семафори с идентификатор semid
- IPC_STAT – получава се информация за масива семафори с semid чрез аргумента arg.buf
- IPC_SET – изменят се някои атрибути на масива семафори и arg.buf определя новите значения (arg.bug.sem_perm.uid, arg.bug.sem_perm.gid, arg.bug.sem_perm.mode)
- GETALL – връща значения на всички елементи в масива семафори чрез аргумента arg.array
- GETVAL – връща значение на семафор с номер semnum като значение на функцията
- SETALL – променя значенията на всички семафори в масива чрез arg.array
- SETVAL – променя значението на семафор с номер semnum на arg.val

Унищожаването на семафор се извършва незабавно, т.е. при изпълнението на функцията semctl. Ако има процеси, които са блокирани в semop, те се събуждар и функцията връща -1 и код EIDRM в errno.