5. УПРАВЛЕНИЕ НА ПРОЦЕСИ

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

5.1. Контекст на процес

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

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

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

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

Пример

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

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

#include "ourhdr.h"

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

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

Ако изпълним програмата ще получим изход от вида:

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

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

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

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

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

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

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

Пример

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

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

#include "ourhdr.h"

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

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

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

Ако изпълним програмата ще получим изход от вида:

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

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

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

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

Пример

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

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

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

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

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

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

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

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

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

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

Ако изпълним програмата ще получим изход от вида:

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

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

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

Съществуват системни примитиви, които връщат значенията на 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.
gid t getegid(void);                   Връща egid на процеса.
```

5.2. Системен примитив `fork`

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

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
                Връща 0 в процеса-син, pid на сина в процеса-баща, -1 при грешка.
```

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

Пример

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

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

#include "ourhdr.h"

main(void)
{
  pid_t pid;

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

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

След `fork` и двата процеса работят асинхронно и не можем да разчитаме кой ще работи първи и кой втори. Затова процесът-баща извиква `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. При изпълнението на програмата получихме следния изход.

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

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

Пример

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

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

#include "ourhdr.h"

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

main(void)
{
  int local;
  pid_t pid;

  local = 77;

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

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

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

При изпълнение на програмата получаваме следния изход.

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

```
write to standard output
before fork
pid = 1842, glob = 6, local = 78
before fork
pid = 1841, glob = 5, local = 77
```

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

Забележете взаимодействието на `fork` с операциите по вход-изход. Тъй като при писане с `write` данните не се буферират, то изходът от `write` преди `fork` се появява веднаж на стандартния изход. Изход чрез функции от стандартната В/И библиотека обаче се буферира. При първото изпълнение на програмата стандартният изход е свързан с терминала и изходът от `printf` се изхвърля от буфера при символа за нов ред (line buffered). При второто изпълнение на програмата стандартният изход е пренасочен към файл `out` и изходът от `printf` се появява два пъти. Това е така защото се прави пълно буфериране на изхода от `printf` (fully buffered). Процесът-син, който е копие на бащата, получава и копие на буфера с данните и второто извикване на `printf` в сина добавя новите данните към получените от бащата. Когато двата процеса завършат съдържанието на буферите им се изхвъля.

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

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

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

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

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

Има два основни начина за използване на `fork`.

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

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

Има няколко начина за завършване на процес:

1. Нормално завършване:
   - при извикване на `exit` или `_exit`
   - при изпълнение на `return` от функцията `main`, което е еквивалентно на извикването на `exit`.
2. Аварийно завършване:
   - при получаване на сигнал, за който реакцията на процеса е завършване (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.
```

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

Пример

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

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

#include "ourhdr.h"

static void my_exit1(void), my_exit2(void);

main(void)
{
  if (atexit(my_exit2) != 0)
     err_sys_exit("atexit my_exit2 error");
  if (atexit(my_exit1) != 0)
     err_sys_exit("atexit my_exit1 error");
  if (atexit(my_exit1) != 0)
     err_sys_exit("atexit my_exit1 error");
```

```
  printf("main is done\n");
  exit(0); /* or return(0); */
}

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

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

При изпълнението на програмата получаваме следния изход.

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

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

5. 4. Системни примитиви `wait`

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

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

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

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

1. Функцията `wait` чака първия син, който завърши. Функцията `waitpid` има аргумент *`pid`*, чрез който може да се чака завършването на определен син. Интерпретацията на аргумента *`pid`* е следната:
   - *`pid`* `== -1` Чака първия син, който завърши.
   - *`pid`* `> 0` Чака син с идентификатор *`pid`*.
   - *`pid`* `== 0` Чака първия син от същата група процеси.
   - *`pid`* `< -1` Чака първия син от група процеси с идентификатор |*`pid`*|.

2. Функцията `wait` винаги блокира процеса, ако синът не е завършил. Чрез аргумента *`options`* на `waitpid` може да се предотврати блокирането на процеса. При значение `WNOHANG`, процесът не се блокира ако синът не е завършил, а функцията връща 0.

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

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

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

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

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

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

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

Пример

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

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

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

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

```
#endif
            );
  else if(WIFSTOPPED(status))
     printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}

main(void)
{
  int status;
  pid_t pid;

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

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

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

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

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

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

При изпълнението на програмата получаваме следния изход.

```
$ a.out
normal termination, exit status = 2
```

```
abnormal termination, signal number = 6 (core file)
abnormal termination, signal number = 15
abnormal termination, signal number = 8 (core file)
abnormal termination, signal number = 14
```

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

Пример

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

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

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

main(void)
{
  pid_t pid;

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

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

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

Ако изпълним програмата ще получим изход от вида.

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

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

Пример

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

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

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

main(void)
{
  pid_t pid;

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

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

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

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

Ако изпълним програмата ще получим изход от вида.

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

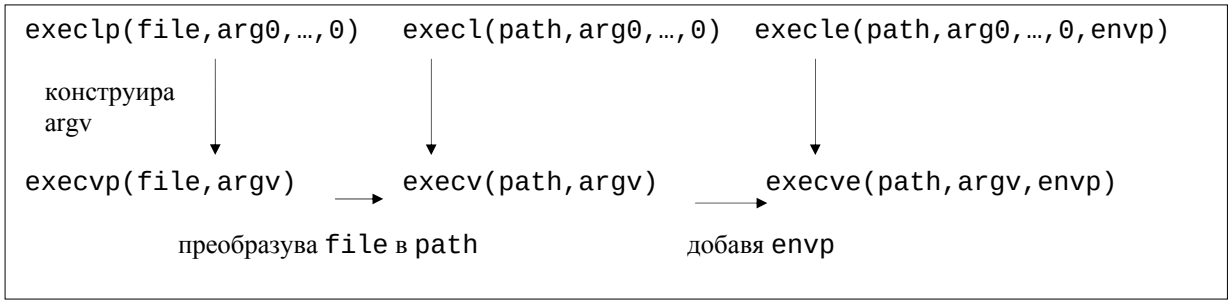
5.5. Системни примитиви `exec`

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

```
#include <unistd.h>

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

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

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

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

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

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

Връща -1 при грешка, нищо при успех.

Връзката между функциите на примитива `exec` е показана на Фиг. 5.2.



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

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

Пример

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

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

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

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

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

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

Ако изпълним програмата ще получим изход от вида.

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

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

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

- променливите от обкръжението, ако се използва функция `execle` или `execve`, в която е указан аргумент `envp`;
- ефективния потребителски идентификатор - ако е вдигнат set-UID бита за файла с име *`name`*;

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

5.6. Потребителски идентификатори на процес

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

```
uid_t getuid(void);
                                                    Връща ruid на процеса.
uid_t geteuid(void);
                                                    Връща euid на процеса.
gid_t getgid(void);
                                                    Връща rgid на процеса.
gid_t getegid(void);
                                                    Връща egid на процеса.
```

Промяна на ефективния идентификатор може да стане по два начина.

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

Пример

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

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

#include <stdio.h>

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

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

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

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

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

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

```
#include <unistd.h>
int setuid(uid_t uid);
```

Връща 0 при успех, -1 при грешка.

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

Пример

Програма 5.12 е илюстрира примитива `setuid`.

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

#include <fcntl.h>

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

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

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

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

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

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

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

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

5.7. Групи процеси и сесия

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

Следват системните примитиви свързани с понятието група процеси.

```
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.

Пример

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

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

#include "ourhdr.h"

main(void)
{
  int pid, status;

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

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

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

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

Като изпълнихме програмата в Scientific Linux получихме следния изход.

```
$ a.out
Parent: pid=16235, grp=16235
Child:  pid=16236, grp=16235
Child after setpgrp: pid=16236, grp=16236
```

```
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     16235 Ss     501   0:00 -bash
12467 16235 16235 12467 tty1     16235 S+     501   0:00 a.out
16235 16236 16236 12467 tty1     16235 S      501   0:00 a.out
16236 16237 16236 12467 tty1     16235 R      501   0:00 ps j
```

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

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

Следват системните примитиви свързани с понятието сесия.

```
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.

Как и кога се свързва управляващ терминал с процес? Обикновено ние не трябва да се тревожим за управляващия терминал, тъй като го получаваме при login. Всеки процес наследява управляващия терминал от процеса-баща при `fork`, а при `exec` управляващият терминал не се променя. Процесът `getty` (`mingetty` в Linux) има управлявляващ терминал, който се наследява от login-shell процеса, който става лидер на сесия. В някои версии на Linux (като Red Hat Linux 4.1, която сме използвали при теста на някои от примерите) лидер на сесия е процесът `login`, а негов син е процесът `bash`, който принадлежи на същата сесия. След това всички процеси, пораждани от login-shell процеса при изпълнение на команди наследяват от него управляващия терминал и идентификатора на сесия, т.е. принадлежат на една сесия и са свързани с един управляващ терминал.

Но как `getty` процесът е получил управляващ терминал? Процес лидер на сесия установява връзка с управляващ терминал. Начинът, по който се установява връзка с управляващ терминал е системно зависим. Например, в UNIX System V и Linux ядрото свързва управляващ терминал с процес, когато той изпълнява примитива `open` на

специален файл за терминал (напр., `fd=open("/dev/tty1", …);`) и ако са изпълнени условията:

- Терминалът в момента не е управляващ терминал на сесия.
- Процесът е лидер на сесия. (в 4.3BSD е различно)

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

Пример

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

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

#include "ourhdr.h"

main(void)
{
  int pid, status;

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

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

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

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

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

Като изпълнихме програмата в Red Hat Linux 4.1 получихме следния изход.

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

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

В Scientific Linux получихме подобен изход.

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

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

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

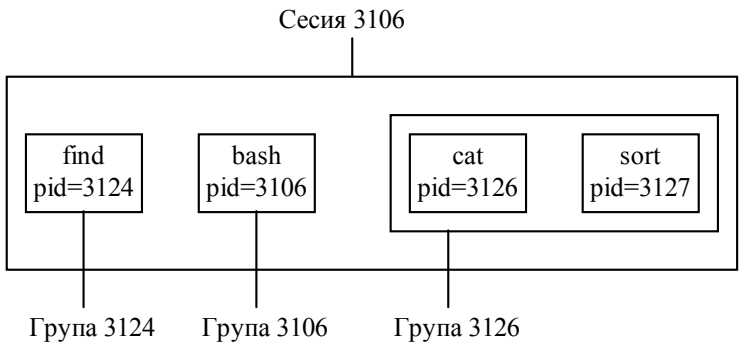
5.8. Управление на заданията

Управление на заданията (Job control) е възможност добавена в BSD към 1980г. и приета в POSIX. Някои командни интерпретатори, като Korn shell, C shell и Bash реализират понятието задание (наричат ги job control shell) като използват групи процеси. Други командни интерпретатори, като B shell не поддържат задания. Заданието е един или повече процеса, които са в една група. Всички процеси пораждани от login-shell процеса са организирани като задания, т.е. групирани в групи, като една от тях е привилегирована, а всички останали са фонови групи. Реализацията на задания изисква подръжка и от ядрото - от терминалния драйвер и механизма на сигналите.

Да разгледаме изпълнението на следващите команди от поддържащ заданията Bash:

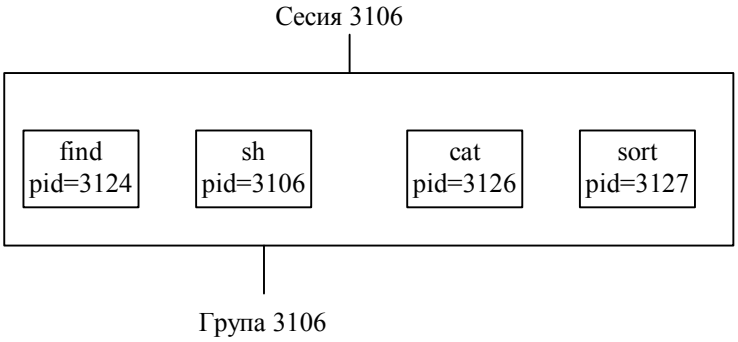
```
$ find /home -name core -print  > outfind 2>&1 &
$ cat | sort
```

Всичките четири процеса `bash`, `find`, `cat` и `sort` принадлежат на една сесия с лидер процеса `bash` (обикновено е така, но може да е и процеса `login`) и имат един и същ управляващ терминал. Процесите `cat` и `sort` принадлежат на една група с лидер `cat`, която в момента е привилегирована, процесът `find` е лидер на друга група и `bash` е лидер на третата група (Фиг.5.3а).



Фиг. 5.3а. Сесия и групи процеси

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



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

Пример

Програма 5.15 ще ни покаже с какъв shell работим.

```
/* ------------------------------------------------------------ */
/* Job control test */

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

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

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

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

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

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

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

Освен това, когато поддържащ заданията shell стартира ново задание в привилегирован режим, сменя полето TPGID в tty структурата на управляващия терминал да съответства на това задание. Докато неподдържащ заданията shell никога не сменя значението на полето TPGID, т.е. това поле винаги съдържа pid на login-shell процеса.

6. СИГНАЛИ

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

6.1. Типове сигнали

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

- Сигнали, свързани с управляващия терминал

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

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

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

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

- Сигнали свързани с апаратни особени ситуации

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

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

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

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

- Сигнали свързани с програмни ситуации

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

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

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

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

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

`SIGKILL` Предизвиква безусловно завършване на процеса.

`SIGTERM` Предупреждение за завършване на процеса.

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

`SIGUSR2` Още един потребителски сигнал.

6.2. Изпращане и обработка на сигнали

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

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

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

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

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

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

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

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

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

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

Когато по-късно пристигне сигнал от съответния тип, той ще бъде обработен според запомнената реакция. Ако това е била потребителска функция, то преди обработката се връща реакцията по премълчаване. Следователно, ако процес иска да обработва повтарящи се сигнали от един тип чрез потребителска функция, трябва отново да изпълнява `signal` след всеки получен сигнал. Това решение изглежда вярно, защото в повечето случаи работи правилно, но не е. Нов сигнал може да пристигне преди процесът да успее да изпълни `signal`, тъй като когато процес работи в потребителска фаза, ядрото може да направи превключване на контекста преди процесът да е стигнал до `signal`. Затова се препоръчва извикването на `signal` да е в началото на функцията, обработваща сигнала (въпреки, че това не решава проблема с повтарящите се сигнали).

Сега вече се виждат недостатъците в семантиката на ненадеждните сигнали:

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

Пример

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

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

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

static void sig_usr(int);

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

  for ( ; ; )
      pause();
}

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

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

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

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

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

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

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

Във всичките случаи е необходимо процесът, изпращащ сигнала, да има права:

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

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

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

```
int pause(void);
```
Връща -1.

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

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

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

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

Пример

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

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

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

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

main(void)
{
  unsigned int ret_sl;

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

  ret_sl = mysleep(4);

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

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

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

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

unsigned int mysleep(unsigned int nsec)
{
  if ( signal(SIGALRM, sig_alrm) == SIG_ERR)
     return(nsec);
  alarm(nsec);          /* start the timer */
  pause();              /* next caught signal wakes up */
  return(alarm(0));     /* turn of timer and return unslept seconds */
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от изпълнението на програмата.

```
$ a.out
```

< въвеждаме <ctrl-c> преди да изтекат 4 сек. >

```
sig_int starting

sig_int finished
mysleep returned 3
$ a.out
```

< изчакваме 4 сек. >

```
mysleep returned 0
```

Пример

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

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

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

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

main(int argc, char *argv[])
{
  struct stat statbuf;
  time_t atime;
  if (argc != 2)
     err_exit("usage: a.out filename");

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

static void wakeup(int sig)
{
  signal(SIGALRM, wakeup);
}

static void quit(int sig)
{
  printf("Termination of process %d\n", getpid());
  exit(0);
}
/* ------------------------------------------------------------ */
```

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

```
$ a.out file1 &
[1] 2234
$ cat file1
```

```
            < съдържанието на файл file1 >
file1: accessed
$ kill 2234
$ Termination of process 2234
$ a.out /no/such/file
stat error: /no/such/file: No such file or directory
```

Пример

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

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

main (void)
{
  int i;

  printf("Parent : pid=%d, group=%d\n", getpid(), getpgrp());
  for (i=0; i<2; i++) {
      if ( fork() == 0 ) {   /* in child processes  */
         if (i & 01 ) setpgrp();
         printf("Child %d: pid=%d, group=%d\n", i, getpid(), getpgrp());
         pause();
       }
  }
  signal(SIGINT, SIG_IGN);    /* without signal kill also parent */
  signal(SIGALRM, wakeup);
  alarm(5);
  pause();
  kill(0, SIGINT);            /* send SIGINT to group */
  printf("Parent after kill\n");
  system("ps j");
  exit(0);
}

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

Следва примерен изход от изпълнението на програмата.

```
$ a.out
Parent : pid=16468, group=16468
Child 0: pid=16469, group=16468
Child 1: pid=16470, group=16470
Parent after kill
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2535 12467 12467 12467 tty1     16468 Ss     501   0:00 -bash
```

```
12467 16468 16468 12467 tty1     16468 R+     501   0:00 a.out
16468 16469 16468 12467 tty1     16468 Z+     501   0:00 [a.out] <defunct>
16468 16470 16470 12467 tty1     16468 S      501   0:00 a.out
16468 16471 16468 12467 tty1     16468 R+     501   0:00 ps j
```

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

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

Пример

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

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

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

void sig_alarm(int);

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

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

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

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

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

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

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

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

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

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

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

Програмата изпълнява следните действията:

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

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

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

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

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