Funkcje jądra systemu operacyjnego UNIX

Dariusz Wawrzyniak

IIPP

29 września 2009

Plan

Plan

Funkcje systemowe

Dostęp do usług jądra systemu operacyjnego UNIX odbywa się poprzez wywołanie odpowiedniej funkcji systemowej. Z punktu widzenie programisty interfejs funkcji systemowych nie różni się niczym od interfejsu funkcji bibliotecznych języka C. Wykonanie funkcji systemowej różni się natomiast od wykonania zwykłej funkcji języka C, gdyż następuje wówczas przełączenie trybu pracy procesora w tzw. *tryb jądra* zwany też *trybem systemowym*.

Wynik funkcji systemowej

Wynik wykonania funkcji systemowej może mieć istotny wpływ na działanie nie tylko procesu wywołującego tę funkcję, ale również na inne procesy działające współbieżnie w systemie. Proces wywołujący otrzymuje jednak najczęściej pewne wartości wynikowe, które są przekazywane przez parametry wyjściowe lub jako wartość zwrotna funkcji. Szczególne znaczenie ma wartość zwrotna –1, oznaczająca, że wykonanie funkcji systemowej zakończyło się błędem. Wartość większa lub równa zero oznacza zakończenie poprawne. Jeśli wartość zwrotna nie ma żadnej semantyki, to w przypadku poprawnego zakończenia jest to wartość 0. W nielicznych przypadkach funkcje systemowe nie zwracają żadnej wartości.

Identyfikacja błędów

W celu stwierdzenie przyczyny wystąpienia błędu w wykonaniu funkcji systemowej po jej zakończeniu, ale przed następnym wywołaniem funkcji systemowej, należy sprawdzić wartość zmiennej globalnej `errno`. Dla użytkownika wartość zmiennej `errno` jest mało czytelna, więc można uzyskać komunikat związany z daną przyczyną błędu.

Komunikat o błędzie

Wygodnym sposobem przekazania komunikatu o błędzie jest użycie funkcji `perror`, której parametrem jest łańcuch znaków definiowany przez programistę i informujący o miejscu wystąpienia błędu, a wynikiem działania jest przekazanie na tzw. standardowe wyjście awaryjne komunikatu złożonego z przekazanego do funkcji łańcucha znaków oraz systemowego komunikatu o przyczynie błędu. Należy zwrócić uwagę, że **wywołanie funkcji `perror` ma sens tylko wówczas, gdy wykonana wcześniej funkcja systemowa zakończyła się błędem**, czyli zwróciła wartość –1.

Przykład

```
#include <fcntl.h>

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

    if ( argc < 2 )
        exit (1);

    d = open(argv[1], O_RDONLY);
    if ( d == -1 ) {
        perror("Blad otwarcia pliku");
        exit (1);
    }

    exit (0);
}
```

Plan

Otwarcie pliku

`int` **`open`**`(const char *`*pathname*`, int` *flags*`)` — otwarcie pliku.

Funkcja zwraca deskryptor otwartego pliku lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
flags	tryb otwarcia:
	`O_WRONLY` — tylko do zapisu,
	`O_RDONLY` — tylko do odczytu,
	`O_RDWR` — do zapisu lub do odczytu.

Odczyt pliku

`ssize_t` **`read`**`(int` *fd*`, void *`*buf*`, size_t` *count*`)` — odczyt z pliku. Funkcja zwraca liczbę odczytanych bajtów, lub –1 w przypadku błędu. Zwrócenie wartości 0 oznacza osiągnięcie końca pliku.

Opis parametrów:

fd deskryptor pliku, z którego następuje odczyt danych,

buf adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane,

count liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane).

Zapis pliku

`ssize_t` **`write`**`(int` *fd*`, const void *`*buf*`, size_t` *count*`)` —
zapis do pliku. Funkcja zwraca liczbę zapisanych bajtów, lub –1 w przypadku błędu.

Opis parametrów:

fd	deskryptor pliku, do którego zapisywane są dane,
buf	adres początku obszaru pamięci, zawierającego blok danych do zapisania w pliku,
count	liczba bajtów do zapisania w pliku (rozmiar zapisywanego bloku).

Przesuwanie wskaźnika bieżącej pozycji

`off_t` **`lseek`**`(int` *fd*`, off_t` *offset*`, int` *whence*`)` —
przesunięcie wskaźnika bieżącej pozycji. Funkcja zwraca wartość wskaźnika bieżącej pozycji po przesunięciu, liczoną względem początku pliku lub –1 w przypadku błędu.

Opis parametrów:

fd	deskryptor otwartego pliku,
offset	wielkość przesunięcia (ujemna — cofanie, dodatnia — w kierunku końca pliku)
whence	odniesienie, może przyjąć jedną z wartości:
	`SEEK_SET` — względem początku pliku,
	`SEEK_END` — względem końca pliku,
	`SEEK_CUR` — względem bieżącej pozycji.

Skracanie pliku

`int` **`truncate`**`(const char *`*pathname*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez nazwę. Funkcja zwraca 0 w przypadku pomyślnego zakończenia lub –1 w przypadku błędu.

`int` **`ftruncate`**`(int` *fd*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez deskryptor. Funkcja zwraca 0 w przypadku pomyślnego zakńczenia lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
fd	deskryptor pliku,
length	rozmiar w bajtach, do którego nastąpi skrócenie.

Zamykanie pliku

`int` **`close`**`(int` *fd*`)` — zamknięcie deskryptora pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor pliku, który zostanie zamknięty.

Tworzenie pliku

`int` **`creat`**`(const char *`*pathname*`, mode_t` *mode*`)` —
utworzenie nowego pliku lub usunięcie jego zawartości, gdy już istnieje oraz otwarcie go do zapisu. Funkcja zwraca deskryptor pliku do zapisu lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
mode	prawa dostępu do nowo tworzonego pliku.

Usuwanie pliku

`int` **`unlink`**`(const char *`*pathname*`)` — usunięcie dowiązania do pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pathname nazwa pliku (w szczególności nazwa ścieżkowa).

Deskryptory

Wszystkie funkcje przydzielające deskryptory (np. **`open`**, **`creat`**) alokują deskryptor o najniższym wolnym numerze. Programista nie ma bezpośredniego wpływu na przydzielony numer deskryptora i w większości przypadków numer ten nie ma istotnego znaczenia.

Standardowe strumienie

We wszystkich programach standardowych zakłada się, że pewne deskryptory odgrywają szczególną rolę, polegającą na identyfikacji standardowych strumieni danych w następujący sposób:

- deskryptor nr 0 — standardowe wejście,
- deskryptor nr 1 — standardowe wyjście,
- deskryptor nr 2 — standardowe wyjście awaryjne.

Powielanie deskryptorów

`int` **`dup`**`(int` *fd*`)` — powielenie (duplikacja) deskryptora. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor, który ma zostać powielony.

`int` **`dup2`**`(int` *oldfd*`,` `int` *newfd*`)` — powielenie (duplikacja) deskryptora we wskazanym miejscu w tablicy deskryptorów. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.

Opis parametrów:

oldfd deskryptor, który ma zostać powielony,

newfd numer nowo przydzielanego deskryptora.

Plan

Identyfikacja procesów

Każdy proces w systemie musi mieć swój unikalny identyfikator, odróżniający go od innych procesów i umożliwiający jednoznaczne wskazanie konkretnego procesu, gdy wymagają tego mechanizmy systemu operacyjnego. W środowisku systemu operacyjnego UNIX powszechnie używanym skrótem terminu *identyfikator procesu* jest PID (ang. Process IDentyfier). Jednym z atrybutów procesu jest też identyfikator jego przodka, określany skrótem PPID.

Identyfikator procesu — PID

PID jest liczbą całkowitą typu `pid_t` (w praktyce `int`). Każdy nowo tworzony proces otrzymuje jako identyfikator kolejną liczbę typu `pid_t`, a po dojściu do końca zakresu liczb danego typu przydział rozpoczyna się od początku. Przydzielane są oczywiście tylko identyfikatory nie używane w danej chwili przez inne procesy.
PID danego procesu nie może ulec zmianie, można go natomiast łatwo uzyskać wywołując funkcję systemową **`getpid`**. Identyfikator przodka można uzyskać za pomocą funkcji **`getppid`**.

Uzyskiwanie identyfikatorów

`pid_t` **`getpid`**`(void)` — uzyskanie własnego identyfikatora (PID) przez proces. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

`pid_t` **`getppid`**`(void)` — uzyskanie identyfikatora procesu macierzystego (PPID) przez potomka. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

Tworzenie procesu

`pid_t` **`fork`**`(void)` — utworzenie procesu potomnego. W procesie macierzystym funkcja zwraca identyfikator (pid) procesu potomnego (wartość większą od 0, w praktyce większą od 1), a w procesie potomnym wartość 0. W przypadku błędu funkcja zwraca –1, a proces potomny nie jest tworzony.

Przykład

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    fork();
    printf("Koniec\n");
}
```

Uruchamianie programu

W ramach istniejącego procesu może nastąpić uruchomienie innego programu wyniku wywołania jednej z funkcji systemowych **`execl`**, **`execlp`**, **`execle`**, **`execv`**, **`execvp`**, **`execve`**. Funkcje te będą określane ogólną nazwą **`exec`**. Uruchomienie nowego programu oznacza w rzeczywistości zmianę programu wykonywanego dotychczas przez proces, czyli zastąpienie wykonywanego programu innym programem, wskazanym odpowiednio w parametrach aktualnych funkcji **`exec`**.

Funkcje typu **exec**

```
int execl(const char *path,const char *arg,...)
int execlp(const char *file,const char *arg,...)
int execle(const char *path,const char *arg ,...,
    char *const envp[])
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execve(const char *file, char *const argv[], char *
    const envp[])
```

Opis parametrów:

path	nazwa ścieżkowa pliku z programem,
file	nazwa pliku z programem,
arg	argument łinii poleceń",
argv	wektor (tablica) argumentów łinii poleceń",
envp	wektor zmiennych środowiskowych.

Przykład

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    execlp("ls", "ls", "-a", NULL);
    printf("Koniec\n");
}
```

Przykład

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    if (fork() == 0){
        execlp("ls", "ls", "-a", NULL);
        perror("Blad uruchmienia programu");
        exit(1);
    }
    printf("Koniec\n");
}
```

Zakończenie procesu

Proces może się zakończyć dwojako: w sposób normalny, tj. przez wywołanie funkcji systemowej **`exit`** lub w sposób awaryjny, czyli przez wywołanie funkcji systemowej **`abort`** lub w wyniku reakcji na sygnał. Funkcja systemowa **`exit`** wywoływana jest niejawnie na końcu wykonywania programu przez proces lub może być wywołana jawnie w każdym innym miejscu programu. Funkcja kończy proces i powoduje przekazanie w odpowiednie miejsce tablicy procesów kodu wyjścia, czyli wartości, która może zostać odebrana i zinterpretowana przez proces macierzysty.

Funkcja **exit**

`void` **`exit`**`(int` *status*`)` — zakończenie procesu.

Opis parametrów:

status kod wyjścia przekazywany procesowi macierzystemu.

Zabicie procesu

Zakończenie procesu w wyniku otrzymania sygnału nazywane jest zabiciem. Proces może otrzymać sygnał wysłany przez jakiś inny proces (również przez samego siebie) za pomocą funkcji systemowej **`kill`** lub wysłany przez jądro systemu operacyjnego.

Uzyskanie informacji o zakończeniu potomka

Proces macierzysty może się dowiedzieć o sposobie zakończenie bezpośredniego potomka przez wywołanie funkcji systemowej **`wait`**. Jeśli wywołanie funkcji **`wait`** nastąpi przed zakończeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie. Po zakończeniu potomka w procesie macierzystym następuje wyjście z funkcji **`wait`**, przy czym pod adresem wskazanym w parametrze aktualnym umieszczony zostanie *status zakończenia*, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo kod wyjścia (bardziej znaczący bajt), przekazany przez potomka jako wartość parametru funkcji **`exit`**. Najbardziej znaczący bit młodszego bajtu wskazuje, czy nastąpił zrzut zawartości pamięci, czyli czy został utworzony plik `core`.

Funkcja `wait`

`pid_t` **`wait`**`(int *`*status*`)` — oczekiwanie na zakończenie potomka.

Funkcja zwraca identyfikator (pid) procesu potomnego, który się zakończył lub –1 w przypadku błędu.

Opis parametrów:

status adres słowa w pamięci, w którym umieszczony zostanie status zakończenia.

Przykład

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    if (fork() == 0){
        execlp("ls", "ls", "-a", NULL);
        perror("Blad uruchmienia programu");
        exit(1);
    }
    wait(NULL);
    printf("Koniec\n");
}
```

Dziedziczenie tablicy deskryptorów

Proces dziedziczy tablicę deskryptorów od swojego przodka. Jeśli nie nastąpi jawne wskazanie zmiany, standardowym wejściem, wyjściem i wyjściem diagnostycznym procesu uruchamianego przez powłokę w wyniku interpretacji polecenia użytkownika jest terminal, gdyż terminal jest też standardowym wejściem, wyjściem i wyjściem diagnostycznym powłoki.

Przeadresowanie standardowych strumieni

Zmiana standardowego wejścia lub wyjścia możliwa jest dzięki temu, że funkcja systemowa **exec** nie zmienia stanu tablicy deskryptorów. Możliwa jest zatem podmiana odpowiednich deskryptorów w procesie przed wywołaniem funkcji **exec**, a następnie zmiana wykonywanego programu. Nowo uruchomiony program w ramach istniejącego procesu zastanie ustawione odpowiednio deskryptory otwartych plików i pobierając dane ze standardowego wejścia (z pliku o deskryptorze 0) lub przekazując dane na standardowe wyjście (do pliku o deskryptorze 1) będzie lokalizował je w miejscach wskazanych jeszcze przed wywołaniem funkcji **exec** w programie.

Zasady przydziału deskryptorów

Jednym ze sposobów zmiany standardowego wejścia, wyjścia lub wyjścia diagnostycznego jest wykorzystanie faktu, że funkcje alokujące deskryptory (między innymi **creat**, **open**) przydzielają zawsze deskryptor o najniższym wolnym numerze.

```
#include <stdio.h>

main(int argc, char* argv[]){
    close(1);
    creat("ls.txt", 0600);
    execvp("ls", argv);
}
```

Sieroty

Nowe pojęcie

Sierota to proces potomny, którego przodek już się zakończył.

Sieroty adoptowane są przez proces systemowy `init` o identyfikatorze 1, tzn. po osieroceniu procesu jego przodkiem staje się proces `init`.

Przykład

```
#include <stdio.h>

main(){
    if (fork() == 0){
        sleep(30);
        exit(0);
    }
    exit(0);
}
```

Zombi

Nowe pojęcie

Zombi to proces potomny, który zakończył swoje działanie i czeka na przekazanie statusu zakończenia przodkowi.

System nie utrzymuje procesów zombi, jeśli przodek ignoruje sygnał `SIGCLD` (`SIGCHLD`).

Przykład

```
#include <stdio.h>

int main(){
    if (fork() == 0)
        exit(0);
    sleep(30);
    wait(NULL);
}
```

Plan

Łącza jako pliki

Łącza w systemie UNIX są plikami specjalnymi, służącymi do komunikacji pomiędzy procesami. Łącza mają kilka cech typowych dla plików zwykłych, czyli posiadają swój i-węzeł, posiadają bloki z danymi (chociaż ograniczoną ich liczbę), na otwartych łączach można wykonywać operacje zapisu i odczytu.

Specyfika łączy

- ograniczona liczba bloków — łącza mają rozmiar 4KB – 8KB w zależności od konkretnego systemu,
- dostęp sekwencyjny — na deskryptorze łącza nie można przemieszczać wskaźnika bieżącej pozycji (nie można wykonywać funkcji **`lseek`**),
- specyfika operacji odczytu — dane odczytane z łącza są zarazem usuwane (nie można ich odczytać ponownie),
- proces jest blokowany w funkcji **`read`** na pustym łączu, jeśli jest otwarty jakiś deskryptor tego łącza do zapisu,
- proces jest blokowany w funkcji **`write`**, jeśli w łączu nie ma wystarczającej ilości wolnego miejsca do zapisania całego bloku,
- przepływ strumienia — dane odczytywane są w kolejności, w której były zapisywane.

Rodzaje łączy

- łącze nazwane (kolejki FIFO) — ma dowiązanie w systemie plików, co oznacza, że jego nazwa jest widoczna w jakimś katalogu i może ona służyć do identyfikacji łącza.
- łącze nienazwane (potoki) — nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza.

Komunikacja przez łącza nienazwane

Ponieważ łącze nienazwane nie ma dowiązania w systemie plików, nie można go identyfikować przez nazwę. Jeśli procesy chcą się komunikować za pomocą takiego łącza, muszą znać jego deskryptory. Oznacza to, że procesy muszą uzyskać deskryptory tego samego łącza, nie znając jego nazwy. Jedynym sposobem przekazania informacji o łączu nienazwanym jest przekazanie jego deskryptorów procesom potomnym dzięki dziedziczeniu tablicy otwartych plików od swojego procesu macierzystego. Za pomocą łącza nienazwanego mogą się zatem komunikować procesy, z których jeden otworzył łącze nienazwane, a następnie utworzył pozostałe komunikujące się procesy, które w ten sposób otrzymają w tablicy otwartych plików deskryptory istniejącego łącza.

Tworzenie łącza nienazwanego

`int` **`pipe`**`(int` *`fd`*`[2])` — utworzenie łącza nienazwanego. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd tablica 2 deskryptorów, która jest parametrem wyjściowym (`fd[0]` jest deskryptorem potoku do odczytu, a `fd[1]` jest deskryptorem potoku do zapisu).

Przykład

```
main() {
   int pdesk[2];

   if (pipe(pdesk) == -1){
      perror("Tworzenie potoku");
      exit(1);
   }

   switch(fork()){
      case -1:    // blad w tworzeniu procesu
         perror("Tworzenie procesu");
         exit(1);
      case 0:     // proces potomny
         if (write(pdesk[1], "Hallo!", 7) == -1){
            perror("Zapis do potoku");
            exit(1);
         }
         exit(0);
      default: {  // proces macierzysty
         char buf[10];
         if (read(pdesk[0], buf, 10) == -1){
            perror("Odczyt z potoku");
            exit(1);
         }
         printf("Odczytano z potoku: %s\n", buf);
      }
   }
}
```

Przykład potoku `ls | tr a-z A-Z`

```
main(int argc, char* argv[]) {
   int pdesk[2];

   if (pipe(pdesk) == -1){
      perror("Tworzenie potoku");
      exit(1);
   }

   switch(fork()){
      case -1: // blad w tworzeniu procesu
         perror("Tworzenie procesu");
         exit(1);
      case 0: // proces potomny
         dup2(pdesk[1], 1);
         execvp("ls", argv);
         perror("Uruchomienie programu ls");
         exit(1);
      default: { // proces macierzysty
         close(pdesk[1]);
         dup2(pdesk[0], 0);
         execlp("tr", "tr", "a-z", "A-Z", 0);
         perror("Uruchomienie programu tr");
         exit(1);
      }
   }
}
```

Komunikacja przez łącza nazwane

Operacje zapisu i odczytu na łączu nazwanym wykonuje się tak samo, jak na łączu nienazwanym, inaczej natomiast się je tworzy i otwiera. Łącze nazwane tworzy się poprzez wywołanie funkcji **`mkfifo`** w programie procesu lub przez wydanie polecenia **`mkfifo`** na terminalu. Funkcja **`mkfifo`** tworzy plik specjalny typu łącze podobnie, jak funkcja **`creat`** tworzy plik zwykły. Funkcja **`mkfifo`** nie otwiera jednak łącza i tym samym nie przydziela deskryptorów. Łącze nazwane otwierane jest funkcją **`open`** podobnie jak plik zwykły, przy czym łącze musi zostać otwarte jednocześnie w trybie do zapisu przez jeden proces i w trybie do odczytu przez inny proces. W przypadku wywołania funkcji **`open`** tylko w jednym z tych trybów proces zostanie zablokowany aż do momentu, gdy inny proces nie wywoła funkcji **`open`** w trybie komplementarnym.

Tworzenie łącza nazwanego

`int` **`mkfifo`**`(const char *`*pathname*`, mode_t` *mode*`)` —
utworzenie pliku specjalnego typu łącze. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pathname nazwa pliku (w szczególności nazwa ścieżkowa),

mode prawa dostępu do nowo tworzonego pliku.

Przykład

```
#include <fcntl.h>

main(){
    mkfifo("kolFIFO", 0600);
    open("kolFIFO", O_RDONLY);
}
```

Plan

Przegląd mechanizmów IPC

	tworzenie *get*	operacje *op*	kontrola *ctl*
pamięć współdzielona	`shmget`	`shmat` `shmdt`	`shmctl`
semafory	`semget`	`semop`	`semctl`
kolejki komunikatów	`msgget`	`msgsnd` `msgrcv`	`msgctl`

Tworzenie obiektu IPC — funkcja *get*

- Pierwszy parametr — *key* (wartość całkowita typu `key_t`) — jest odnośnikiem do konkretnego obiektu w ramach danego rodzaju mechanizmów lub stałą `IPC_PRIVATE`.
- Ostatni parametr — *flg* — określa prawa dostępu do nowo tworzonego obiektu reprezentującego mechanizm IPC opcjonalnie połączone (sumowane bitowo) z flagami:
  - `IPC_CREAT` — w celu utworzenia obiektu, jeśli nie istnieje,
  - `IPC_EXCL` — w celu wymuszenia zgłoszenia błędu, gdy obiekt ma być utworzony, ale już istnieje.
- Wartość zwrotna — identyfikator, na który odwzorowywany jest klucz lub –1 w przypadku błędu.

Operacje na obiekcie IPC — *op*

- Pierwszy parametr — *id* — identyfikator zwrócony przez odpowiednią funkcję *get*.
- Pozostałe parametry zależne są od rodzaju mechanizmu i specyfiki operacji.
- Wartość zwrotna — w przypadku poprawnego wykonania wartość większa lub równa 0, zależnie do wykonywanego polecenia; –1 w przypadku błędu.

Operacje kontrolne na obiekcie IPC — *ctl*

- Pierwszy parametr — najczęściej *id* — identyfikator zwrócony przez odpowiednią funkcję *get*.
- Drugi parametr — *cmd* — stała określająca rodzaj operacji:
  - `IPC_RMID` — usunięcie obiektu IPC,
  - `IPC_STAT` — odczyt atrybutów obiektu IPC (identyfikator właściciela, grupy, prawa dostępu itp.),
  - `IPC_SET` — modyfikacja atrybutów obiektu IPC.
- Wartość zwrotna — w przypadku poprawnego wykonania wartość większa lub równa 0, zależnie do wykonywanego polecenia; –1 w przypadku błędu.

Tworzenie segmentu pamięci współdzielonej

`int` **`shmget`**`(key_t` *key*`, int` *size*`, int` *shmflg*`)` — utworzenie segmentu pamięci współdzielonej. Funkcja zwraca identyfikator segmentu pamięci współdzielonej w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

key	klucz,
size	rozmiar obszaru współdzielonej pamięci w bajtach,
shmflg	flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Przyłączenie segmentu pamięci współdzielonej

`void *`**`shmat`**`(int` *shmid*`, const void *`*shmaddr*`, int` *shmflg*`)`

— włączenie segmentu współdzielonej pamięci w przestrzeń adresową procesu. Funkcja zwraca adres segmentu lub –1 w przypadku błędu.

Opis parametrów:

shmid	identyfikator obszaru współdzielonej pamięci, zwrócony przez funkcję **`shmget`**,
shmaddr	adres w przestrzeni adresowej procesu, pod którym ma być dostępny segment współdzielonej pamięci (wartość `NULL` oznacza wybór adresu przez system),
shmflg	flagi, specyfikujące sposób przyłączenia (np. `SHM_RDONLY` — przyłączenie tylko do odczytu).

Odłączenie segmentu pamięci współdzielonej

`int` **`shmdt`**`(const void *`*shmaddr*`)` — wyłączenie segmentu z przestrzeni adresowej procesu. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

shmaddr adres początku obszaru współdzielonej pamięci w przestrzeni adresowej procesu.

Przykład utworzenia obszaru pamięci współdzielonej

```
int *buf;

shmid = shmget(45281, MAX*sizeof(int), IPC_CREAT|0600);
if (shmid == -1){
   perror("Utworzenie segmentu pamieci wspoldzielonej");
   exit(1);
}

buf = (int*)shmat(shmid, NULL, 0);
if (buf == NULL){
   perror("Przylaczenie segmentu pamieci wspoldzielonej");
   exit(1);
}
```

Tworzenie tablicy semaforów

`int` **`semget`**`(key_t` *key*`, int` *nsems*`, int` *semflg*`)` — utworzenie zbioru (tablicy) semaforów. Funkcja zwraca identyfikator zbioru semaforów w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

key	klucz,
nsems	liczba semaforów w tworzonym zbiorze,
semflg	flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Wykonanie operacji na tablicy semaforów

`int` **`semop`**`(int` *semid*`, struct sembuf *`*sops*`, unsigned` *nsops*`)` — wykonanie operacji semaforowej. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

semid identyfikator zbioru semaforów, zwrócony przez funkcję **`semget`**,

sops adres tablicy struktur, w której każdy element opisuje operację na jednym semaforze w zbiorze,

nsops rozmiar tablicy adresowanej przez *sops* (liczba elementów o strukturze `sembuf`).

Struktura opisująca operację na semaforze

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

Znaczenie poszczególnych pól:

sem_num numer semafora w zbiorze,

sem_op wartość dodawana do zmiennej semaforowej,

sem_flg flagi operacji (`IPC_NOWAIT` — wykonanie bez blokowania, `SEM_UNDO` — wycofanie operacji w przypadku zakończenia procesu).

Struktura opisująca operację na semaforze

`int` **`semctl`**`(int` *semid*`, int` *semnum*`, int` *cmd*`, ...)` — wykonanie operacji kontrolnych na semaforze lub zbiorze semaforów. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

semid	identyfikator zbioru semaforów, zwrócony przez funkcję **`semget`**,
semnum	numer semafora, na którym ma być wykonana operacja, w identyfikowanym zbiorze,
cmd	specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `IPC_RMID`, `SETVAL`, `GETVAL`, `SETALL` itp.).

Operacja opuszczania semafora ogólnego

```
static struct sembuf buf;

void opusc(int semid, int semnum){
    buf.sem_num = semnum;
    buf.sem_op = -1;
    buf.sem_flg = 0;
    if (semop(semid, &buf, 1) == -1){
        perror("Opuszczenie semafora");
        exit(1);
    }
}
```

Operacja podnoszenia semafora ogólnego

```
static struct sembuf buf;

void podnies(int semid, int semnum){
    buf.sem_num = semnum;
    buf.sem_op = 1;
    buf.sem_flg = 0;
    if (semop(semid, &buf, 1) == -1){
        perror("Podnoszenie semafora");
        exit(1);
    }
}
```

Definiowanie wartości zmiennej semaforowej

```
if (semctl(semid, 0, SETVAL, (int)MAX) == -1){
   perror("Nadanie wartosci semaforowi 0");
   exit(1);
}
if (semctl(semid, 1, SETVAL, (int)0) == -1){
   perror("Nadanie wartosci semaforowi 1");
   exit(1);
}
```

Tworzenie kolejki komunikatów

`int` **`msgget`**`(key_t` *key*`, int` *msgflg*`)` — utworzenie kolejki komunikatów. Funkcja zwraca identyfikator kolejki w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

key klucz,

msgflg flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Wysyłanie komunikatu

`int` **`msgsnd`**`(int` *msgid*`, struct msgbuf *`*msgp*`, size_t` *msgsz*`, int` *msgflg*`)` — wysyłanie komunikatu o zawartości wskazywanej przez *msgp* z ewentualnym blokowaniem procesu w przypadku braku miejsca w kolejce. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

msgid	identyfikator kolejki komunikatów,
msgp	wskaźnik na bufor z treścią i typem komunikatu do wysłania,
msgsz	rozmiar fragmentu bufora, zawierającego właściwą treść komunikatu,
msgflg	flagi (np. `IPC_NOWAIT`).

Struktura komunikatu

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Znaczenie poszczególnych pól:

mtype typ komunikatu (wartość większa od 0),

mtext treść komunikatu,

Odbiór komunikatu

`ssize_t` **`msgrcv`**`(int` *msgid*`, struct msgbuf *`*msgp*`, size_t` *msgsz*`, long` *msgtyp*`, int` *msgflg*`)` — odbiór komunikatu z ewentualnym blokowaniem procesu w przypadku braku w kolejce komunikatu, spełniającego wymagane parametry. Funkcja zwraca rozmiar treści odczytanej od pola *mtext* lub –1 w przypadku błędu.

Opis parametrów:

msgid	identyfikator kolejki komunikatów,
msgp	wskaźnik na bufor,
msgsz	rozmiar fragmentu bufora do przechowania właściwej treści komunikatu,
msgtyp	typ komunikatu,
msgflg	flagi (np. `IPC_NOWAIT`, `MSG_NOERROR`, `MSG_EXCEPT`).

Plan

Sygnały

Sygnał wysyłany jest do procesu w celu zakomunikowania o fakcie wystąpienia pewnego zdarzenia istotnego dla tego procesu. Sygnał może być wysłany przez jakiś proces w systemie (w szczególności proces może wysłać sygnał sam do siebie) lub przez jądro systemu operacyjnego. Możliwość przesłanie sygnału przez proces do innego procesu uwarunkowana jest zgodnością odpowiednich identyfikatorów. W przypadku zwykłych użytkowników obowiązujący identyfikator użytkownika procesu wysyłającego musi być taki sam, jak rzeczywisty identyfikator użytkownika procesu, do którego adresowany jest sygnał. Procesy użytkownika uprzywilejowanego `root` mogą wysyłać sygnały do wszystkich procesów w systemie.

Rodzaje sygnałów (1)

nazwa	nr	reakcja	opis
SIGHUP	1	zakończenie	zawieszenie terminala
SIGINT	2	zakończenie	naciśnięcie klaw. przerw.
SIGQUIT	3	zrzut obrazu	naciśniecie klaw. zakończ.
SIGILL	4	zrzut obrazu	nieprawidłowa instrukcja
SIGABRT	6	zrzut obrazu	wywołanie funkcji **`abort`**
SIGFPE	8	zrzut obrazu	wyjątek zmiennopozycyjny
SIGKILL	9	zakończenie	sygnał zabicia procesu
SIGSEGV	11	zrzut obrazu	błąd odnieś do pamięci
SIGPIPE	13	zakończenie	przerwanie potoku
SIGALRM	14	zakończenie	alarm od czasom. (**`alarm`**)

Rodzaje sygnałów (2)

nazwa	numer	reakcja	opis
SIGTERM	15	zakończ.	sygnał zakończenia
SIGUSR1	30, 10, 16	zakończ.	sygnał 1 użytkownika
SIGUSR2	31, 12, 17	zakończ.	sygnał 2 użytkownika
SIGCHLD	20, 17, 18	ignor.	zakończenie potomka
SIGCONT	19, 18, 25		kontynuacja po zatrzym.
SIGSTOP	17, 19, 23	zatrzym.	zawieszenie procesu
SIGTSTP	18, 20, 24	zatrzym.	sygnał zatrzym. z term.
SIGTTIN	21, 21, 26	zatrzym.	odczyt z term. w tle
SIGTTOU	22, 22, 27	zatrzym.	zapis na term. w tle

Wysłanie sygnału do innego procesu

`int` **`kill`**`(pid_t` *pid*`,` `int` *signum*`)` — wysłanie sygnału do procesu. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pid	identyfikator procesu, do którego adresowany jest sygnał,
signum	numer przesyłanego sygnału.

Wysłanie sygnału „do siebie”

W celu wysłania sygnału do samego siebie proces może użyć funkcji systemowej **`raise`**. Wywołanie tej funkcji jest funkcjonalnie równoważne wywołaniu funkcji **`kill`** z wynikiem wywołania funkcji **`getpid`** jako pierwszym argumentem. W sposób szczególny proces może wysłać do samego siebie sygnał `SIGALRM`. Wysłanie tego sygnału jest następstwem wywołania funkcji **`alarm`**.

Reakcja na sygnał

- reakcja domyślna,
- ignorowanie sygnału,
- przechwytywanie sygnału — zdefiniowanie reakcji na sygnał

Reakcję na sygnał można zdefiniować, korzystając z funkcji **`signal`** lub **`sigaction`**.

Definiowanie reakcji na sygnał

`int` **`signal`**`(int` *signum*`, void (*`*handler*`)(int))` —
zdefiniowanie reakcji procesu na sygnał.

Opis parametrów:

signum numer przesyłanego sygnału,

handler wskaźnik na funkcję do obsługi sygnału lub jedna ze stałych:

`SIG_DFL` — ustawienie reakcji domyślnej,

`SIG_IGN` — ignorowanie sygnału.

Przykład definiowania reakcji na sygnał

```
void f(int sig_num){
   printf("Przechwycenie sygnalu nr %d\n", sig_num);
}

main(){

   printf("Domyslna obsluga sygnalu\n");
   signal(SIGINT, SIG_DFL);
   sleep(5);
   printf("Ignorowanie sygnalu\n");
   signal(SIGINT, SIG_IGN);
   sleep(5);
   printf("Przechwytywanie sygnalu\n");
   signal(SIGINT, f);
   sleep(5);

}
```