12. Komunikacja między procesami

Wstęp

Procesy działające współbieżnie w systemie wielozadaniowym mogą współpracować lub być niezależne od siebie. Współpracujące procesy wymagają od systemu dostarczenia mechanizmów, które umożliwią im komunikowanie się ze sobą oraz synchronizowanie swoich działań.

Wykład 12 zawiera omówienie podstawowych zagadnień dotyczących komunikacji między procesami. Prezentujemy kolejno najważniejsze mechanizmy oferowane przez system Linux: łącza komunikacyjne, kolejki komunikatów i pamięć dzieloną. Omawiamy sposób ich implementacji oraz podstawowe operacje dostępne poprzez funkcje systemowe.

12.1. Podstawowe zagadnienia i mechanizmy komunikacji

Istnieją dwa podstawowe schematy komunikowania się procesów:

- pamięć dzielona,
- system komunikatów.

W schemacie pamięci dzielonej kilka procesów może wspólnie użytkować ten sam obszar pamięci. Procesy wymieniają informacje poprzez bezpośrednie zapisywanie i odczytywanie danych z tej pamięci. Z tego względu jest to najszybsza metoda komunikowania sie procesów. Na samych procesach, a właściwie na twórcach programów, spoczywa obowiązek zorganizowania komunikacji. Zadaniem systemu operacyjnego jest tylko umożliwienie współdzielenia pewnych obszarów pamięci.

W systemie komunikatów procesy mogą wymieniać informacje w postaci komunikatów. System operacyjny zapewnia mechanizm przesyłania komunikatów pomiędzy procesami. W tym celu muszą być zdefiniowane następujące elementy:

- łącze komunikacyjne,
- operacje: **nadaj**(komunikat) i **odbierz**(komunikat).

Obydwa schematy komunikacji uzupełniają się wzajemnie i są często wykorzystywane jednocześnie w systemach operacyjnych. Niektóre systemy, takie jak Mach czy Windows NT, w dużym zakresie opierały swoje działanie na przesyłaniu komunikatów i wykorzystywały ten mechanizm nawet do komunikowania się niektórych modułów systemu.

W dalszej części niniejszej lekcji przedstawione zostaną przykładowe implementacje obydwu powyższych schematów zastosowane w systemie Linux.

System Linux udostępnia następujące mechanizmy synchronizacji i komunikacji procesów:

- sygnały,
- pliki,
- łącza komunikacyjne:
    - łącza nienazwane,
    - łącza nazwane,
- mechanizmy IPC Systemu V:
    - semafory,
    - kolejki komunikatów,
    - pamięć dzielona,
- gniazda.

Wykorzystanie sygnałów do komunikowania się procesów zostało już omówione w niniejszym podręczniku. Stosowne informacje można odnaleźć w wykładach 4 i 8. Korzystaniu z plików poświęcone są wykłady 3 i 11. Wykorzystanie gniazd w komunikacji sieciowej procesów opisano w wykładzie 14.

12.2. Łącza nienazwane

Łącze nienazwane umożliwia jednokierunkową komunikację pomiędzy procesami.

Jeden z procesów wysyła dane do łącza a drugi odczytuje te dane w kolejności, w jakiej zostały wysłane. Łącze ma więc organizację kolejki FIFO (ang. First In First Out) i ograniczoną pojemność.

Łącza realizowane są jako obiekty tymczasowe w pamięci jądra i udostępniane poprzez interfejs systemu plików. Każde łącze reprezentowane jest przez strukturę danych zwaną i-węzłem, podobnie jak każdy plik w systemie. Różnica polega na tym, że tymczasowy i-węzeł łącza wskazuje stronę w pamięci a nie bloki dyskowe.

Tworząc łącze na zlecenie procesu, jądro otwiera je od razu do czytania i pisania oraz dodaje dwie nowe pozycje do tablicy deskryptorów plików otwartych w procesie. Procesy nie posługują się więc w ogóle nazwą łącza i stąd pochodzi określenie "**łącze nienazwane**". Z tego też powodu korzystać z łącza mogą wyłącznie procesy spokrewnione, czyli proces macierzysty i kolejne pokolenia procesów potomnych. Procesy potomne dziedziczą po procesie macierzystym deskryptory wszystkich otwartych plików, w tym deskryptory otwartych łączy, co umożliwia im korzystanie z łączy utworzonych przez proces macierzysty.

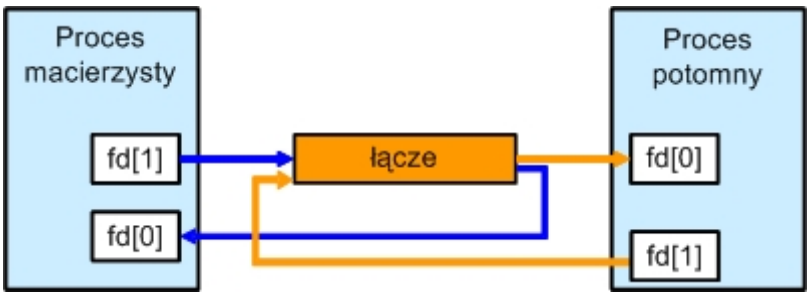
Funkcja systemowa **pipe**() tworzy łącze nienazwane i otwiera je zarówno do czytania jak i do pisania.

```
int pipe(int fd[2]);
```

Funkcja zwraca tablicę dwóch deskryptorów: **fd**[0] umożliwiający czytanie z łącza i **fd**[1] umożliwiający pisanie do łącza. Ponieważ dwa procesy mogą się komunikować przez łącze tylko w jedną stronę, więc żaden z nich nie wykorzysta obydwu deskryptorów. Każdy z procesów powinien zamknąć nieużywany deskryptor łącza za pomocą funkcji **close**().

```
int close(int fd);
```

Jeden z procesów zamyka łącze do czytania a drugi do pisania. Uzyskuje się wtedy jednokierunkowe połączenie między dwoma procesami, co ilustruje rysunek 12.1.



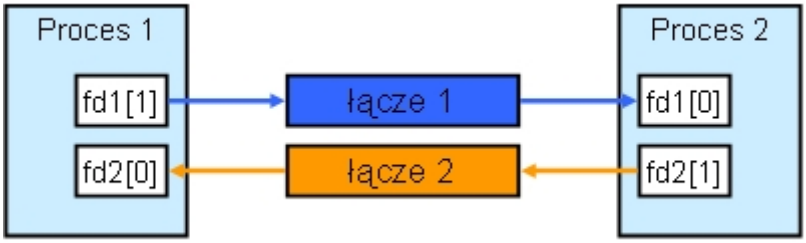
Rys. 12.1 Zastosowanie łącza nienazwanego do jednokierunkowej komunikacji między procesami

W podobny sposób interpreter poleceń realizuje przetwarzanie potokowe. W celu wykonania złożonego polecenia:

```
ps -ef | more
```

powłoka tworzy łącze komunikacyjne i dwa procesy potomne, które wykonują programy **ps** i **more**.

Dwukierunkowa komunikacja między procesami wymaga użycia dwóch łączy komunikacyjnych. Jeden z procesów pisze do pierwszego łącza i czyta z drugiego, a drugi proces postępuje odwrotnie. Obydwa procesy zamykają nieużywane deskryptory. Sytuację taką przedstawia rysunek 12.2.



Rys. 12.2 Zastosowanie łączy nienazwanych do dwukierunkowej komunikacji między procesami

Do czytania i pisania można użyć funkcji systemowych **read**() i **write**().

```
ssize_t read(int fd, void *buf, size_t count);
```

Funkcja **read** wczytuje **count** bajtów z łącza o deskryptorze **fd** do bufora **buf** i zwraca liczbę wczytanych bajtów. Jeżeli łącze jest puste lub brakuje w nim odpowiedniej porcji danych, to funkcja blokuje proces do momentu pojawienia się danych.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Funkcja **write** zapisuje **count** bajtów z bufora **buf** do łącza o deskryptorze **fd** i zwraca liczbę zapisanych bajtów. Jeżeli liczba bajtów nie przekracza pojemności łącza, to jądro gwarantuje niepodzielność zapisu danych do łącza. W przeciwnym przypadku dane będą zapisane w kilku porcjach i może nastąpić ich przemieszanie, jeśli z łącza korzysta jednocześnie kilka procesów piszących. Jeżeli łącze jest przepełnione, to funkcja blokuje proces w oczekiwaniu na zwolnienie miejsca.

Ustawienie flagi O_NDELAY zmienia działanie obydwu funkcji na nieblokujące i powoduje natychmiastowy powrót z błędem, gdy operacja nie może być zrealizowana. W celu ustawienia flagi trzeba wykorzystać funkcję systemową **fcntl**(), ponieważ proces nie używa jawnie funkcji **open**() do otwarcia łącza.

Przykład
Prezentujemy poniżej kod programu demonstrujący komunikację pomiędzy procesem macierzystym i potomnym z wykorzystaniem łącz nienazwanych.

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

int main(void)
{
  int fd1[2], fd2[2];
  pid_t pid, ppid;
  char bufor[10];

  printf("Zglasza sie proces macierzysty.\n");
  if (pipe(fd1) == -1) {
    perror("pipe");
    exit(1);
  }
  else if (pipe(fd2) == -1) {
    perror("pipe");
    exit(1);
  }
  if ((pid = fork()) == -1) {
    perror("fork");
```

```
    exit(1);
  }
  if (pid == 0) {
    printf("Zglasza sie proces potomny.\n");
    close(fd1[1]);
    close(fd2[0]);
    pid = getpid();
    write(fd2[1], &pid, sizeof(pid));
    read(fd1[0], &ppid, sizeof(pid));
    printf("Proces potomny (%d):
    Identyfikator procesu macierzystego PPID = %d\n", pid, ppid);
    exit(0);
  }
  else {
    close(fd1[0]);
    close(fd2[1]);
    ppid = getpid();
    write(fd1[1], &ppid, sizeof(pid));
    read(fd2[0], &pid, sizeof(pid));
    printf("Proces macierzysty (%d):
            Identyfikator procesu potomnego PID = %d\n", ppid, pid);
  }
  return(0);
}
```

12.3. Łącza nazwane

Łącza nazwane realizowane jest przez system jako pliki typu FIFO. Dzięki temu umożliwiają komunikację między dowolnymi procesami.

Łącze nazwane można utworzyć posługując się funkcją systemową **mknod**(). Funkcja ta służy do tworzenia plików specjalnych (plików urządzeń) oraz plików FIFO (łączy nazwanych):

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

gdzie:

pathname - nazwa ścieżkowa tworzonego pliku,

mode - tryb pliku, definiujący typ i prawa dostępu do pliku,

dev - numery urządzenia, główny i drugorzędny.

Tryb pliku podaje się jako sumę bitową stałej określającej typ tworzonego pliku oraz praw dostępu zapisanych w kodzie ósemkowym.

Argument **dev** nie ma znaczenia podczas tworzeniu łącza nazwanego.

Wywołanie funkcji może więc wyglądać następująco:

```
mknod("/tmp/fifo", S_IFIFO|0666, 0);
```

Z funkcji **mknod**() korzystają dwa polecenia systemowe umożliwiające utworzenie łącza z poziomu interpretera poleceń:

```
mkfifo [opcje] plik

mknod [opcje] plik typ
```

gdzie:

plik - nazwa ścieżkowa tworzonego pliku,

typ - typ pliku: p (FIFO), b, c.

Po utworzeniu, łącze należy otworzyć do czytania bądź pisania. Można w tym celu skorzystać z funkcji systemowej **open**() lub funkcji **fopen**() z biblioteki standardowej języka C.

```
int open(const char *pathname, int flags, mode_t mode);
```

Domyślnie otwarcie łącza jest operacją blokującą. Proces jest wstrzymywany do momentu otwarcia łącza przez inny proces do komplementarnej operacji w stosunku do czytania bądź pisania. Sytuacja taka nie wystąpi, jeżeli proces otwiera łącze jednocześnie do czytania i pisania, jak to ma miejsce w przypadku łączy nienazwanych. Ustawienie flag O_NDELAY lub O_NONBLOCK w wywołaniu funkcji powoduje, że otwarcie oraz wszystkie inne operacje na deskryptorze pliku FIFO stają się nieblokujące. W przypadku braku drugiego procesu funkcja **open**() zwraca wtedy błąd. Wspomniane flagi można też ustawić funkcją **fcntl**().

Operacje czytania i pisania do łącza można zrealizować za pomocą funkcji systemowych **read**() i **write**() albo za pomocą licznych funkcji wejścia/wyjścia z biblioteki standardowej języka C, w zależności od sposobu otwarcia łącza. Domyślnie wszystkie operacje są blokujące, podobnie jak dla łączy nienazwanych.

Zamykanie łącza, podobnie jak każdego innego pliku, odbywa się za pomocą funkcji **close**() lub **fclose**() w zależności od sposobu otwarcia.

W przeciwieństwie do łączy nienazwanych, pliki FIFO pozostają w systemie plików po zakończeniu ich używania przez wszystkie procesy. Dopiero jawne wywołanie funkcji **unlink**() powoduje usunięcie łącza nazwanego.

```
int unlink(const char *pathname);
```

Przykład
Prezentujemy dwa proste programy: klient i serwer, pokazujące sposób utworzenia i wykorzystania FIFO. Serwer tworzy FIFO, otwiera je do czytania i oczekuje na dane. Kod programu przedstawiono poniżej.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>

#define FIFO_FILE "MYFIFO"

int main(void)
{
  FILE *fp;
  char readbuf[80];

/*tworzenie FIFO*/
  umask(0);
  mknod(FIFO_FILE, S_IFIFO|0666, 0);

  while(1)
  {
    if ((fp = fopen(FIFO_FILE, "r"))==NULL) {
      perror("fopen");
      exit(1);
    }
    if (fgets(readbuf, 80, fp)==NULL) {
      perror("fgets");
      exit(1);
    }
    printf("Orzymany tekst: %s\n", readbuf);
    fclose(fp);
  }

  return(0);
}
```

Program klienta próbuje otworzyć FIFO i zapisać do niego dane.

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE "MYFIFO"

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

  if ( argc != 2 ) {
```

```
    printf("Wywolanie: fifoclient [napis]\n");
    exit(1);
  }

  if((fp = fopen(FIFO_FILE, "w")) == NULL) {
    perror("fopen");
    exit(1);
  }

  if (fputs(argv[1], fp)==EOF) {
    perror("fputs");
    exit(1);
        }

  fclose(fp);
  return(0);
}
```

12.4. Mechanizmy IPC Systemu V

W wersji UNIX-a System V wprowadzono trzy nowe mechanizmy komunikacji międzyprocesowej:

1. semafory,
2. kolejki komunikatów,
3. pamięć dzieloną albo wspólną.

Obecnie większość implementacji systemu UNIX oraz Linux udostępnia te mechanizmy. Są one powszechnie określane wspólną nazwą "komunikacja międzyprocesowa Systemu V" lub w skrócie IPC (ang. System V Interprocess Communication).

Implementacja

W ramach każdego mechanizmu jądro tworzy pojedyncze obiekty na zlecenie procesów. Każdy obiekt reprezentowany jest przez oddzielną strukturę danych. Dla każdego z mechanizmów jądro przechowuje tablicę wskaźników na struktury poszczególnych obiektów.

Tablica 9.1 Mechanizmy i obiekty IPC

Mechanizm	Obiekt	Reprezentacja
semafory	zbiór semaforów	struktura **semid_ds**
kolejki komunikatów	kolejka komunikatów	struktura **msqid_ds**
pamięć dzielona	segment pamięci dzielonej	struktura **shmid_ds**

Do utworzenia obiektu potrzebny jest unikalny **klucz** w postaci 32-bitowej liczby całkowitej. Klucz ten stanowi nazwę obiektu, która jednoznacznie go identyfikuje i pozwala procesom uzyskać dostęp do utworzonego obiektu. Każdy obiekt otrzymuje również swój identyfikator, ale jest on unikalny tylko w ramach jednego mechanizmu. Oznacza to, że może istnieć kolejka i zbiór semaforów o tym samym identyfikatorze.

Wartość klucza można ustawić dowolnie. Zalecane jest jednak używanie funkcji **ftok**() do generowania wartości kluczy. Nie gwarantuje ona wprawdzie unikalności klucza, ale znacząco zwiększa takie prawdopodobieństwo.

```
key_t ftok(char *pathname, char proj);
```

gdzie:

pathname - nazwa ścieżkowa pliku,

proj - jednoliterowy identyfikator projektu.

Wszystkie tworzone obiekty IPC mają ustalane prawa dostępu na podobnych zasadach jak w przypadku plików. Prawa te ustawiane są w strukturze **ipc_perm** niezależnie dla każdego obiektu IPC.

Obiekty IPC pozostają w pamięci jądra systemu do momentu, gdy:

- jeden z procesów zleci jądru usunięcie obiektu z pamięci,
- nastąpi zamknięcie systemu.

Operacje

Wszystkie trzy mechanizmy korzystają z podobnych funkcji systemowych, zestawionych w tablicy 9.2.

Tablica 9.2 Funkcje systemowe operujące na obiektach IPC

Obiekt / Typ operacji	zbiór semaforów	kolejka komunikatów	segment pamięci dzielonej
tworzenie i otwieranie	semget()	msgget()	shmget()
operacje sterujące	semctl()	msgctl()	shmctl()
operacje specyficzne	semop()	msgsnd() msgrcv()	shmat() shmdt()

Polecenia systemowe

Polecenie **ipcs** wyświetla informacje o wszystkich obiektach IPC istniejących w systemie, dokonując przy tym podziału na poszczególne mechanizmy. Wyświetlane informacje obejmują m.in. klucz, identyfikator obiektu, nazwę właściciela, prawa dostępu.

```
ipcs [ -asmq ] [ -tclup ]

ipcs [ -smq ] -i id
```

Wybór konkretnego mechanizmu umożliwiają opcje:

-s - semafory,

-m - pamięć dzielona,

-q - kolejki komunikatów,

-a - wszystkie mechanizmy (ustawienie domyślne).

Dodatkowo można podać identyfikator pojedyńczego obiektu **-i id**, aby otrzymać informacje tylko o nim.

Pozostale opcje specyfikują format wyświetlanych informacji.

Dowolny obiekt IPC można usunąć posługując się poleceniem:

```
ipcrm [ shm | msg | sem ] id
```

gdzie:

shm, msg, sem - specyfikacja mechanizmu, kolejno: pamięć dzielona, kolejka komunikatów, semafory,

id - identyfikator obiektu.

12.5. Kolejki komunikatów

Utworzenie nowej kolejki komunikatów lub otwarcie dostępu do już istniejącej umożliwia funkcja systemowa **msgget**().

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

gdzie:

key - klucz,

msgflg - flagi.

Funkcja zwraca identyfikator kolejki związanej z podaną wartością klucza **key**. Szczegółowy sposób działania wynika z flag ustawionych w argumencie **msgflg**:

0		- funkcja otwiera istniejącą kolejkę z podanym kluczem lub zwraca błąd, jeśli kolejka nie istnieje,
IPC_CREAT		- funkcja tworzy nową kolejkę lub otwiera istniejącą kolejkę z podanym kluczem,
IPC_EXCL	**|**	- funkcja tworzy nową kolejkę lub zwraca błąd, jeśli kolejka z podanym
IPC_CREAT		kluczem już istnieje.

Argument **msgflg** może również opcjonalnie zawierać maskę praw dostępu do kolejki. Podawany jest wtedy jako suma bitowa stałych symbolicznych określających flagi oraz maski praw dostępu w kodzie ósemkowym.

Operacje przesyłania komunikatów wymagają posłużenia się buforem komunikatu zdefiniowanym w następujący sposób:

```
struct msgbuf {

    long mtype;       // typ komunikatu (wartość > 0)

    char mtext[1];    // tekst komunikatu

};
```

Pole **mtype** określa typ komunikatu w postaci dodatniej liczby całkowitej. Tablica **mtext**[] przechowuje treść komunikatu, którą mogą stanowić dowolne dane. Rozmiar tablicy podany w definicji nie stanowi rzeczywistego ograniczenia, ponieważ bufor komunikatu można dowolnie przedefiniować w programie pod warunkiem zachowania typu na początku.

Funkcja **msgsnd**() umożliwia przesłanie komunikatu do kolejki:

```
int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
```

gdzie:

msqid - identyfikator kolejki komunikatów,

msgp - wskaźnik do bufora zawierającego komunikat do wysłania,

msgsz - rozmiar bufora komunikatu z wyłączeniem typu (rozmiar treści komunikatu),

msgflg - flagi.

Funkcja **msgrcv**() pobiera z kolejki jeden komunikat wskazanego typu. Pobrany komunikat jest usuwaney z kolejki.

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtype,
int msgflg);
```

gdzie:

msqid - identyfikator kolejki komunikatów,

msgp - wskaźnik do bufora, do którego ma być zapisany komunikat, pobrany z kolejki,

msgsz - rozmiar bufora komunikatu z wyłączeniem typu (rozmiar treści komunikatu),

msgtype - typ komunikatu do pobrania z kolejki,

msgflg - flagi.

Argument **msgtype** specyfikuje typ komunikatu do pobrania w następujący sposób:

msgtype > 0 - pobiera najstarszy komunikat danego typu,

msgtype = 0 - pobiera najstarszy komunikat w kolejce.

Obydwie opisane operacje są domyślnie operacjami blokującymi proces do momentu pomyślnego zakończenia. Ustawienie flagi IPC_NOWAIT w wywołaniu funkcji zmienia jej działanie na nieblokujące.

Funkcja sterująca **msgctl**() umożliwia pobranie lub ustawienie atrybutów kolejki, jak również usunięcie kolejki ze struktur danych jądra.

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

gdzie:

msqmid - identyfikator kolejki,

cmd - operacja sterująca,

buf - wskaźnik do bufora przeznaczonego na strukturę **msqid_ds** kolejki.

Argument **cmd** decyduje o rodzaju operacji sterującej wykonywanej na kolejce:

IPC_STAT - powoduje zapisanie zawartości struktury **msqid_ds** kolejki do bufora **buf**,

IPC_SET - powoduje ustawienie w strukturze **ipc_perm** praw dostępu do kolejki pobranych z bufora,

IPC_RMID - usuwa kolejkę komunikatów z jądra.

Przykład

W pliku **msg_snd.c** zamieszczono kod programu, który wysyła jeden komunikat do kolejki. Typ oraz treść komunikatu podawane są w linii wywołania programu.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

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

int main(int argc, char *argv[])
{
        key_t   key;
        int     msgflg, msqid;
        struct msgbuf {
                long mtype;
                char mtext[80];
        } message;

        if (argc!=3) {
            printf("Wywołanie: %s typ wiadomosc\n", argv[0]);
            exit(1);
        }
        key=ftok(".", 'c');
        msgflg=0666|IPC_CREAT;

        if( (msqid = msgget(key,msgflg)) == -1)
        {
                perror("msgget");
                exit(1);
        }

        message.mtype = atoi(argv[1]);
        strcpy(message.mtext, argv[2]);

        if ( msgsnd(msqid, &message, 80, 0) < 0)
        {
                perror("msgsnd");
                exit(1);
        }

}
```

W pliku **msg_rcv.c** zamieszczono kod programu, który odbiera z kolejki jeden komunikat typu podanego jako argument wywołania. Jeżeli argument zostanie pominięty, to program odbierze najstarszy komunikat w kolejce.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        key_t   key;
        int     msgflg, msqid;
        long    type;
        struct msgbuf {
                long mtype;
                char mtext[80];
        } message;
```

```
        if (argc==2)
                type=atoi(argv[1]);
        else if (argc==1)
                type=0;
        else {
                printf("Poprawne wywołanie: %s typ\n", argv[0]);
                exit(1);
        }

        key=ftok(".", 'c');
        msgflg=0;

        if( (msqid=msgget(key,msgflg)) == -1)
        {
                perror("msgget");
                exit(1);
        }

        if (msgrcv(msqid, (struct msgbuf*)&message, 80, type, 0) < 0)
        {
                perror("msgrcv");
                exit(1);
        }
        printf("Odebrana wiadomosc typu %d:  %s\n",
                message.mtype, message.mtext);

}
```

12.6. Pamięć dzielona

Utworzenie nowego segmentu pamięci dzielonej lub uzyskanie dostępu do już istniejącego umożliwia funkcja systemowa **shmget**().

```
int shmget(key_t key, int size, int shmflg);
```

gdzie:

key - klucz,

size - rozmiar segmentu pamięci,

shmflg - flagi.

Funkcja zwraca identyfikator segmentu pamięci związanego z podaną wartością klucza **key**. Szczegółowy sposób działania wynika z flag ustawionych w argumencie **shmflg**:

0 - funkcja udostępnia isniejący segment z podanym kluczem lub zwraca błąd, jeśli segment nie istnieje,

IPC_CREAT - funkcja tworzy nowy segment lub udostępnia isniejący segment z podanym kluczem,

IPC_EXCL | IPC_CREAT - funkcja tworzy nowy segment lub zwraca błąd, jeśli segment z podanym kluczem już istnieje.

Argument **shmflg** może również opcjonalnie zawierać maskę praw dostępu do segmentu pamięci. Podawany jest wtedy jako suma bitowa stałych symbolicznych określających flagi oraz maski praw dostępu w kodzie ósemkowym.

W wynika wywołania funkcji **shmget**() proces uzyskuje identyfikator segmentu pamięci dzielonej. Aby można było z niego korzystać, segment musi zostać jeszcze przyłączony do wirtualnej przestrzeni adresowej procesu za pomocą funkcji **shmat**().

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

gdzie:

shmid - identyfikator segmentu pamięci dzielonej,

shmaddr - adres w przestrzeni adresowj procesu, od którego ma być dołączony segment,

shmflg - flagi.

Funkcja zwraca adres początkowy dołączonego segmentu w wirtualnej przestrzeni adresowej procesu. Adres ten może być wyspecyfikowany w argumencie **shmaddr**. Jądro systemu próbuje wtedy dołączyć segment od podanego adresu pod warunkiem, że jest on wielokrotnością rozmiaru strony pamięci. Zaokrąglonego adresu w dół do granicy strony może być dokonane przez jądro, jeśli ustawiona jest flaga SHM_RND. Zalecane jest jednak ustawienie shmaddr = 0 w wywołaniu funkcji, aby pozwolić na wybór adresu przez jądro.

Domyślnie segment dołączany jest do czytania i pisania przez proces. Ustawienie flagi SHM_RDONLY powoduje dołączenie segmentu wyłącznie do czytania. W obydwu przypadkach proces musi posiadać odpowiednie uprawnienia do wykonywania wspomnianych operacji.

Po zakończeniu korzystania z segmentu pamięci dzielonej, proces powinien odłączyć go za pomocą funkcji systemowej **shmdt**().

```
int shmdt(const void *shmaddr);
```

gdzie:

shmaddr - adres początkowy segmentu w przestrzeni adresowej procesu.

Odłączenie segmentu nie oznacza automatycznie usunięcia z jądra systemu. Segment pozostaje w pamięci i może być ponownie dołączany przez procesy. W celu usunięcia segmentu trzeba posłużyć się funkcją systemową **shmctl**().

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

gdzie:

shmid - identyfikator segmentu,

cmd - operacja sterująca,

buf - wskaźnik do bufora przeznaczonego na strukturę **shmid_ds** segmentu.

Argument **cmd** decyduje o rodzaju operacji sterującej wykonywanej na segmencie pamięci:

IPC_STAT - powoduje zapisanie zawartości struktury **shmid_ds** segmentu do bufora **buf**,

IPC_SET - powoduje ustawienie w strukturze **ipc_perm** praw dostępu do segmentu pobranych z bufora,

IPC_RMID - zaznacza segment do usunięcia z pamięci.

Polecenie IPC_RMID powoduje jedynie zaznaczenie, że segment ma być usunięty z pamięci, gdy przestanie być używany. Usunięcie segmentu nastąpi dopiero wtedy, gdy wszystkie procesy odłączą go od swoich przestrzeni adresowych.

Przykład

W pliku **shmwrite.c** zamieszczono kod programu, który zapisuje jeden komunikat do pamięci dzielonej. Treść komunikatu podawana jest w linii wywołania programu.

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

#define SEGSIZE 100

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

      if (argc != 2) {
            printf("Poprawne wywołanie: %s wiadomosc\n", argv[0]);
            exit(1);
      }

      key = ftok(".", 'p');
      if ((shmid = shmget(key, SEGSIZE, IPC_CREAT|0666)) == -1)
      {
            perror("shmget");
```

```
            exit(1);
      }

      if ((adres = shmat(shmid, 0, 0)) == -1)
      {
            perror("shmat");
            exit(1);
      }

      strcpy(adres, argv[1]);

      if (shmdt(adres) == -1)
      {
            perror("shmdt");
            exit(1);
      }
}
```

W pliku **shmread.c** zamieszczono kod programu, który odczytuje komunikat z pamięci dzielonej i wypisuje go na standardowym wyjściu (stdout).

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

#define SEGSIZE 100

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

      key = ftok(".", 'p');
      if ((shmid = shmget(key, SEGSIZE, 0)) == -1)
      {
            perror("shmget");
            exit(1);
      }

      if ((adres = shmat(shmid, 0, 0)) == -1)
      {
            perror("shmat");
            exit(1);
      }

      printf("%s\n", adres);

      if (shmdt(adres) == -1)
      {
            perror("shmdt");
            exit(1);
      }
}
```

Bibliografia

[1] Silberschatz A., Galvin P.B.: Podstawy systemów operacyjnych, WNT 2000

[2] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 12.6, 13.6)

[3] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdziały: 10.4, 10.6)

[4] Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdziały: 6, 7.1-7.5, 7.12-7.13)

Nowe pojęcia, definicje i wyrażenia

Termin	Objaśnienie
łącze nazwane	mechanizm umożliwiający jednokierunkową komunikację pomiędzy dowolnymi procesami
łącze nienazwane	mechanizm umożliwiający jednokierunkową komunikację pomiędzy procesami spokrewnionymi
IPC	mechanizmy komunikacji międzyprocesowej, obejmujące semafory, pamięć dzieloną i kolejki komunikatów; istnieją dwie popularne realizacje: IPC Systemu V oraz IPC POSIX
kolejka komunikatów	mechanizm umożliwiający dwukierunkową komunikację pomiędzy dowolnymi procesami poprzez przesyłanie komunikatów
pamięć dzielona	obszar pamięci użytkowany wspólnie przez kilka procesów

Zadania do wykładu 12

Zadanie 1

Napisz program, w którym proces serwera pełni rolę pośrednika w komunikacji asynchronicznej między dwoma procesami klientów za pomocą łączy nazwanych. Serwer odbiera dane od każdego z klientów i przesyła je do drugiego klienta. Kolejność i częstotliwość nadawania przez klientów nie jest określona. Rozwiązanie powinno angażować jak najmniej zasobów systemowych i odznaczać się krótkim czasem reakcji.

Zadanie 2

Napisać program do komunikacji poprzez pamięć dzieloną. Po uruchomieniu program tworzy segment pamięci dzielonej lub uzyskuje dostęp do już istniejącego segmentu. Pamięć ta będzie wykorzystywana przez procesy jako tablica łańcuchów znaków o ustalonym rozmiarze. Następnie proces oczekuje na wpisanie jednego z poleceń na stdin:

- **write k "komunikat"** - zapisz komunikat w wierszu **k** tablicy w pamięci dzielonej,
- **read k** - odczytaj komunikat z wiersza **k** tablicy w pamięci dzielonej.

Zadanie 3

Napisać zestaw programów umożliwiających wymianę krótkich wiadomości tekstowych przez grupę kilku użytkowników. Zaproponować sposób identyfikacji poszczególnych użytkowników. Wykorzystać jedną kolejkę komunikatów IPC.