ŁĄCZA NIENAZWANE

Rodzina systemów **POSIX** zaopatrzona została w mechanizm tworzenie międzyprocesowych łączy komunikacyjnych, zwanych potokami:

- **nienazwanymi** a więc istniejącymi wyłącznie w pamięci jądra obiektów tymczasowych, tworzonymi obok otwieranych w momencie inicjowania procesu standardowymi

	<unistd.h>	<stdio.h>	<iostram>
input	STDIN_FILENO	FILE *stdin	std::cin
output	STDOUT_FILENO	FILE *stdout	std::cout
error	STDERR_FILENO	FILE *stderr	std::cerr
			std::clog

- **nazwanymi**, czyli posiadających dowiązanie w systemie plików, do których odwołania następując **explicite** przez nazwę a czas ich istnienia nie jest ograniczony czasem wykonania procesu.

Każdemu z tych strumieni system operacyjny przypisuje deskryptor pliku, który stanowi unikalną liczbą całkowitą 0, 1, 2, 3, itd. choć trzy pierwsze są przypisane standardowym strumieniom wejściowemu, wyjściowemu i błędu.

Procesy potomne dziedziczą deskryptory łączy po procesach macierzystych.

Liczba możliwych do wykorzystania przez proces deskryptorów, a więc i ilość otwartych plików jest ograniczona.

KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

W gruncie rzeczy można byłoby się zawahać na ile w przypadku problematyki współbieżności, czy – rozpatrywanej tu wieloprocesowości – potrzebne są takie rozwiązania. W końcu, każdy proces potomny stanowi dokładną kopię procesu macierzystego, czy w takim razie nie łatwiej uzyskać komunikację międzyprocesową poprzez zdefiniowane już w obrębie kodu zmiennej (chociażby globalnej). Okazuje się jednak że całość nie przedstawia się aż tak prosto.

Rozważmy hipotetyczną sytuację, że w procesie głównym – z pewnego powodu – zaistniała potrzeba obliczania całek.

Załóżmy, że w rozpatrywanym konkretnie przypadku jest to całka

$$\mathcal{I} = \int_0^1 \sin(2\pi x) \cdot \exp^{-x} dx$$

Gdyby obliczyć jej wartość na drodze analitycznej, to otrzymamy

$$\mathcal{I} = 2\frac{\pi\left(1 - e^{-1}\right)}{1 + 4\pi^2} = 0.09811971024$$

Równocześnie może pojawić się sytuacja, że wyrażenie podcałkowe stanowić będą także i inne funkcje. W zawiązku z czym, logicznym rozwiązaniem byłoby uzupełniania kodu programu o fragment umożliwiający obliczenie wartości całki, drogą kwadratury numerycznej.

KACZA NIENAZWANE

Postać jej deklaracji nagłówkowej funkcji obliczającej kwadraturę można by przyjąć następująco

```
double quad(    unsigned int n,double a,double b,
                double (*fun)(double) );
```

gdzie:

`unsigned int n`	ilość węzłów kwadratury
`double a, b`	granice całkowania
`double fun()`	wskazanie do funkcji stanowiącej wyrażenie podcałkowe

Istnieje wiele skutecznych metod obliczania kwadratur numerycznych, gdyby przyjąć tutaj przykładowo metodę trapezów, to

$$\int_a^b f(x)dx \approx \frac{b-a}{2n}\left(f(x_0)+f(x_1)\right)+$$
$$\frac{b-a}{2n}\left(f(x_1)+f(x_2)\right)+\frac{b-a}{2n}\left(f(x_2)+f(x_3)\right)+$$
$$\ldots\ldots$$
$$\frac{b-a}{2n}\left(f(x_{n-2})+f(x_{n-1})\right)+\frac{b-a}{2n}\left(f(x_{n-1}+f(x_n)\right)$$

$$x_0 = a, x_n = b$$
$$x_i = a + \frac{b-a}{n}\cdot k, k = 0, 1, 2, \ldots, n$$

KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

KACZA NIENAZWANE

Samą funkcję kwadratury numerycznej możemy więc zdefiniować jak niżej.

```
double quad( unsigned int n, double a, double b,
             double (*fun)(double) )
{
    unsigned int k;
    double xk, sum;

    sum = fun( a ) + fun( b );
    for( k=1;k<n;k++ )
    {
        xk = a+ (b-a)*k/n;
        sum += 2.0*fun( xk );
    }

    return ( (b-a)/(2.0*n)*sum );
}
```

Ponieważ proces macierzysty ma (lub może mieć) również i wiele innych zadań do wykonania, obliczenia wartości będą przekazane do utworzonego w tym celu procesu potomnego.

ŁĄCZA NIENAZWANE

Kod źródłowy całości może przedstawiać się następująco.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <math.h>


int main( void )
{
    int pid,status;

    double sine( double );
    double quad( unsigned int,double,double,double (*)(double));
    unsigned int n;
    double a,b,I;
    // Inicjujemy wartości początkowe zmiennych
    a=0.0; b=1.0;
    I=0.0;
    // i jeszcze ilość węzłów kwadratury
    n=3200;
```

KACZA NIENAZWANE

Wywołaniem funkcji **fork()** duplikujemy bieżący proces i różnicujemy kod.

```
    switch( (int)fork() )
    {
        case -1:
            perror( "<!> błąd inicjacji potomka" ); exit( 1 ); break;
        case  0:
            // Obliczamy całkę, ale już w potomku
            I = quad( n,a,b,sine );
            // Dla pewności wyprowadzamy informację o tym co wyliczyliśmy
            printf( "[%d] wartość całki\t%16.6f\n",(int)getpid(),I );
            //... i kończymy działanie potomka
            exit( 0 );
        // Teraz kod dla procesu nadrzędnego
        default:
            // Powiedzmy, że coś tutaj ważnego się dzieje
            printf( "[%d] wykonuje ważne rzeczy...\n",(int)getpid() );
            // Oczekiwanie na wynik z potomka
            pid = (int)wait( &status );
            // ... i mamy gotowy rezultat
            printf( "[%d] zakończył z kodem %d\n",pid,status );
            printf( "[%d] otrzymał wartość\t%16.6f\n",(int)getpid(),I );
    }
    // Na tym program kończy działanie
    return 0;
}
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

Oczywiście w kodzie musimy uzupełnić jeszcze definicję funkcji podcałkowej – przedstawia się ona bardzo elementarnie

```
double sine( double x ){ return sin( 2*M_PI*x )*exp(-x); }
```

Kompilacja i konsolidacja

```
$ gcc -Wall integra.c -o integra -lm
```

zwróćmy uwagę na dyrektywę **-lm** nakazującą konsolidację z biblioteką matematyczną **libm.so** – a efekt wykonania

```
[12401] wartość całki            0.098120
[12400] wykonuje, ważne rzeczy...
[12401] zakończył z kodem 0
[12400] otrzymał wartość         0.000000
```

a więc nieco zaskakujący.

Mimo, że utworzony proces potomny wywiązał się z zadania dobrze uzyskując bardzo dobre przybliżenie

0.098120

wobec wartości dokładnej (wyliczonej na drodze analitycznej)

0.09811971024

jednak proces nadrzędny tej wartości nie otrzymał, w jego przypadku wartość zmiennej **I**, pozostała równa wartości inicjującej.

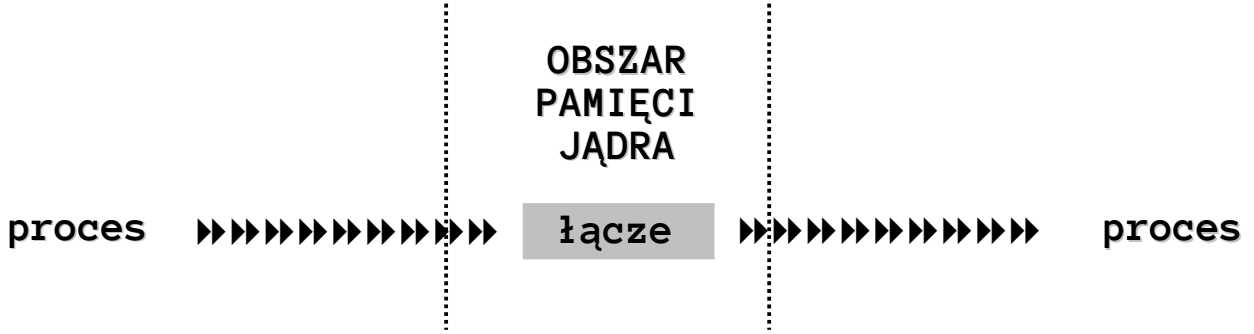
Stało się tak, ponieważ każdy proces posiadał własną, prywatną, kopię zmiennej, stąd i ich wartości końcowe są różne. Natomiast sama komunikacja międzyprocesowa nie przedstawia się aż tak elementarnie i wymaga specjalnych mechanizmów.

ŁĄCZA NIENAZWANE

Wróćmy w takim razie do kwestii użycia łączy – na początek nienazwanych – które mogą stanowić użyteczne narzędzia rozwiązania problemu komunikacji międzyprocesowej.

Umożliwiają one asynchroniczną wymianę danych między pokrewnymi procesami a więc mającymi wspólnego przodka, bądź między bezpośrednio miedzy przodkiem a potomkiem (i odwrotnie).

Realizowane są jako **obiekty tymczasowe w obszarze pamięci jądra**, w ten sposób że dodawane są nowe pozycje do tablicy deskryptorów otwartych plików.



W obrębie danego łącza przepływ odbywa się zawsze w jednym kierunku (**half duplex**), w konsekwencji proces piszący musi – na wstępie – zamknąć odczyt a czytający zapis.

Jeżeli komunikacja międzyprocesowa wymaga także kierunku zwrotnego, to konieczne jest otwarcie dwóch łączy.

Łącza nienazwane nie są objęte standardem ISO/ANSI C lecz POSIX. De facto łącza nienazwane **pipe** stanowią wczesną postać UNIX System IPC (omawianego w dalszej części).

ŁĄCZA NIENAZWANE

Komunikacja międzyprocesowa za pośrednictwem potoków może być realizowana zarówno z poziomu kodu źródłowego **jak i z poziomu powłoki systemowej**, a obecne było w systemach UNIX'opodobnych praktycznie od momentu ich powstania.

Realizowane jest za pośrednictwem operatora przetwarzania potokowego, który łączy standardowe wyjście jednego procesu ze standardowym wejściem innego. Przykładowo

```
$ ls -l /usr/bin/ | cat -n | tail -n 5
2386 -rwxr-xr-x 1 root root   3652 lis 25  2006 znew
2387 lrwxrwxrwx 1 root root      8 kwi  6  2007 zsh -> /bin/zsh
2388 -rwxr-xr-x 1 root root  27920 kwi 18  2007 zsoelim
2389 -rwxr-xr-x 1 root root  10904 lis 27  2006 zvbi-chains
2390 -rwxr-xr-x 1 root root 396688 lip  4  2008 zypper
```

Użyto tutaj trzech instrukcji systemowych:

`ls -l /usr/bin` listing katalogu **`/usr/bin`** w formacie **`long`**
`cat -n` wyświetla zawartość wejście dodając numerację linii
`tail -n 5` wyświetla **`5`** ostatnich linii ze strumienia na wejściu

tworząc, między ich standardowymi wyjściami a wejściami, dwa łącza nienazwane (symbol **`'|'`**).

KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

Korzystając z API systemowego POSIX otwarcie łącza następuje wywołaniem funkcji **`pipe()`** zaś zamknięcie **`close()`**.

Synopsis

```
#include <unistd.h>
int pipe( int fd[2] );
```

`int fd[0]` zawiera numer deskryptora do odczytu

`int fd[1]` zawiera numer deskryptora do zapisu

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

`int fd` numer deskryptora do zamknięcia

Return

`0` operacja zakończona powodzeniem

Errors

`-1`

Odczyt/zapis z/do potoku odbywa się za pośrednictwem pary funkcji **`read()`** i **`write()`**.

Synopsis

```
#include <unistd.h>
ssize_t read( int fd, void *buf, size_t count );
ssize_t write( int fd, const void *buf, size_t count );
```

`void* buf` obszar pamięci do odczytu/zapisu

`size_t count` ilość bajtów do odczytu/zapisu

Return

ilość bajtów przeczytana / zapisana

Errors

`-1`

ŁĄCZA NIENAZWANE

Jako elementarną ilustrację przygotujemy program, który utworzy proces potomny, wysyłający do macierzystego zwrotną wiadomość poprzez takie właśnie łącze.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/limits.h>
int main( void )
{
    int fd[2],n;
    // Definiując bufor wymiany danych użyjemy stałej systemowej PIPE_BUF
    char line[PIPE_BUF];
    // Próbujemy ustanowić łącze (nienazwane)

    if( pipe( fd )< 0 )
    { printf( "...błąd otwarcia łącza\n" ); exit( 1 ); }

    // Tworzymy proces potomny
    switch( fork() )
    {
        // Na wypadek ewentualnego niepowodzenia
        case -1:
            perror( "<!> błąd inicjacji potomka" );
            exit( 1 );
            break;
```

ŁĄCZA NIENAZWANE

Jeżeli **fork()** zakończył się sukcesem, to:

- dla procesu potomnego (**fork()** zwrócił **0**)

```
case  0:
// Na początek zamykamy kanał od strony odczytu
close( fd[0] );
// bo potomek będzie pisał
write( fd[1],"\n\t[pozdrowienia od potomka]\n\n",29 );
// Po wykonaniu zapisu zamykamy (choć właściwie jest to tutaj zbędne)
close( fd[1] );
// ponieważ po wykonaniu exit() system i tak zamknie
// łącze od strony potomka
exit( 0 );
```

- natomiast kod dla procesu nadrzędnego (**fork()** zwrócił **pid**)

```
default:
// Tu natomiast zamykamy zapis (bo po tej stronie jest odczyt)
close( fd[1] );
// Czytamy teraz to co wcześniej wysłał potomek
n = read( fd[0],(void*)line,PIPE_BUF );
close( fd[0] );
// Wypisujemy co otrzymaliśmy od potomka na stdout
write( STDOUT_FILENO,line,n );
}
}
return 0;
}
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

Zauważmy, że celem wyprowadzenia informacji wykorzystano tutaj jeden ze standardowo zdefiniowanych deskryptorów

```
#define STDOUT_FILENO   1       /* Standard output.  */
#define STDERR_FILENO   2       /* Standard error output.  */
```

które otwierane są dla każdego procesu, w momencie jego tworzenia, przez system.
Definicje znajdują się w pliku **unistd.h**.

Rozmiar bufora określono za pomocą stałej **PIPE_BUF** zdefiniowanej w

/linux/limits.h

Jej wartość określa maksymalny rozmiar dla przesyłanej porcji danych poprze łącze nienazwane, tak **aby operacja ta była atomowa** a więc niepodzielna (co w kontekście współbieżności ma znaczenie podstawowe !). Zgodnie z aktualną definicją

```
#define PIPE_BUF        4096
```

Nota bene, obok niej istnieje także definicja (**posix1_lih.h**), o identycznym sensie

```
#define _POSIX_PIPE_BUF 512
```

choć innej (znacznie mniejszej) wartości.

Inaczej niż zwykle, w kodzie procesu macierzystego nie użyto funkcji **wait()**.
Było to możliwe ponieważ wywołanie **read() ma charakter blokujący**. Domniemanie to można zmienić wykorzystując funkcję **fcntl()** i ustawiając znacznik **O_NONBLOC** dla uzyskanego wcześniej deskryptora. Wówczas użycie funkcji **wait()** byłoby konieczne.

ŁĄCZA NIENAZWANE

Wracając teraz do problemu przedstawionego na wstępie – potomka który wykonywał na rzecz procesu nadrzędnego obliczenia (całki) – kod programu powinien się przedstawiać jak w listingu.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <math.h>

int main( void )
{
   int pid,status;

   double sine( double );
   double quad( unsigned int, double, double,double (*)(double) );
   unsigned int n;
   double a,b,I;

   int fd[2];

   a=0.0,b=1.0,I=0.0;
   n=3200;

   if(pipe(fd)<0){ printf("...błąd otwarcia łącza\n"); exit(1); }
```

ŁĄCZA NIENAZWANE

Teraz dopiero tworzymy proces potomny.

```
switch( fork() )
{
    case -1:
        perror( "<!> błąd inicjacji potomka" ); exit( 1 ); break;
    case  0:
        close( fd[0] );
        I = quad( n,a,b,sine );
        printf( "[%d] wartość całki %19.6f\n",(int)getpid(),I );
        write( fd[1],(void*)&I,sizeof( double ) );
        exit( 0 );
    default:
        close( fd[1] );
        printf( "[%d] wykonuje, ważne rzeczy...\n",(int)getpid() );
        read( fd[0],(void*)&I,sizeof( double ) );
        printf( "[%d] zakończył z kodem %d\n",pid,status );
        printf( "[%d] otrzymał wartość %16.6f\n",(int)getpid(),I );
    }

    return 0;
}
```

Oczywiście funkcje **sine()** i **quad()** nie wymagają żadnych zmian (choć są niezbędne).

Efekt wykonania będzie następujący

```
$ integral
[4225] wartość całki             0.098120
[4224] wykonuje, ważne rzeczy...
[4225] zakończył z kodem 0
[4224] otrzymał wartość          0.098120
```

a więc komunikacja między procesami przebiegła bez zakłóceń – a wynik otrzymany przez proces nadrzędny poprawny.

ŁĄCZA NIENAZWANE

Ponieważ proces może utworzyć wiele potoków w postaci łączy nienazwanych, a z reguły w większości implementacji, są one jednokierunkowe, bardzo użytecznymi mogą okazać się funkcje **dup()** i **dup2()**.

Wywołanie obu funkcji utworzy kopie deskryptora **old**, z tą różnicą że w przypadku **dup2()** mamy możliwość wskazania w sposób jawny na co skopiować.
Funkcja **dup()** zwraca natomiast najniższy, pierwszy wolny.

Synopsis

```
#include <unistd.h>

int dup( int old );
int dup2( int old, int new );
```

int old istniejący, utworzony uprzednio wywołaniem **pipe()** deskryptor
int new nowy deskryptor na który zostanie skopiowany (skojarzony) **old**

Return

nowy deskryptor, dla już istniejącego

Errors

-1

KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

W kolejnym przykładzie z procesu macierzystego wyślemy kilka linii tekstu, które w procesie potomnym zostaną posortowane za pomocą programu **sort** (systemowy). Przy okazji użyjemy tu w relacji do łącza nienazwanego funkcji przeznaczonych do działania na strumieniu skojarzonym z plikiem.

```
// duplicate.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
    int fd[2];
    int pid,status;
    // Ta zmienna zostanie w przyszłości skojarzona z potokiem
    FILE* stream;

    if( pipe( fd )< 0 )
    {
        printf( "...błąd otwarcia łącza\n" ); exit( 1 );
    }
```

ŁĄCZA NIENAZWANE

Tworzymy proces potomny

```
switch( (pid=(int)fork()) )
{
    case -1:
        perror( "<!> błąd inicjacji potomka" );
        exit( 1 ); break;
    // Kod dla potomka
    case  0:
        // Na początek powitanie
        printf( "<!>\tpotomek [%d] startuje\n",(int)getpid() );
        // Zamykamy fd[1] bo potomek nie będzie pisał do potoku
        close( fd[1] );
        // Kopjujemy potomkowi fd[0] potoku na jego stdin
        dup2( fd[0],STDIN_FILENO );
        // Zamykamy fd[0],bo już niepotrzebne-skopiowaliśmy na stdin
        close(fd[0]);
        // Teraz pozostaje już tylko wywołać program sort
        printf("----------------------------------------\n" );
        execl( "/usr/bin/sort","sort","--reverse",(char*)NULL );
```

! Program uruchomiony przez **execl()** odziedziczył wejście **stdin** po potomku, a więc przekierowane **fd[0]**.

KACZA NIENAZWANE

Teraz kod dla procesu macierzystego.

```
    default:
        // Zamykamy od tej strony kanału odczyt, bo będziemy pisać.
        close( fd[0] );
        // Przypisanie strumienia (plikowego) istniejącemu deskryptorowi.
        stream = fdopen( fd[1], "w"  );
        // Piszemy do kanału, na końcu którego jest potomek
        // (właściwie to jest tam już systemowe sort).
        fprintf( stream, "\tAaaaa\n" );
        fprintf( stream, "\tBbbbb\n" );
        fprintf( stream, "\tCcccc\n" );
        fprintf( stream, "\tDdddd\n" );
        // Na wszelki wypadek opróżniamy bufor plikowy
        fflush( stream );
        // Zamykamy deskryptor, ponieważ nie jest dłużej potrzebny
        close( fd[1] );
        // i czekamy na potomka, aż skończy
        wait( &status );
        // Po zakończeniu wyświetlamy komunikat
        pintf("----------------------------------------\n" );
        printf("<!>\tpotomek [%d] zakończył działanie i
            zwrócił [%d]\n",pid,status );
    }
    return 0;
}
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

KACZA NIENAZWANE

Kompilacja źródeł

```
$ gcc duplicate.c -o duplicate
```

a efektem wykonania jest

```
$ ./duplicate
<!>     potomek [8269] startuje
-------------------------------------------------------
    Ddddd
    Ccccc
    Bbbbb
    Aaaaa
-------------------------------------------------------
<!>     potomek [8269] zakończył działanie i zwrócił [0]
$
```

Zauważmy, że celem skojarzenia deskryptora ze strumieniem użyto tutaj dodatkowo funkcji **fdopen()** i **fflush()**.

```
#include <stdio.h>
FILE *fdopen( int fildes, const char *mode );
int fflush( FILE *stream );
```

gdzie tryb **mode** może być określony jako:
r (odczyt),
r+|w+ (odczyt lub zapis),
a (rozszerzenie, czyli pisanie na końcu pliku)
a+ (rozszerzenie lub czytanie).

ŁĄCZA NIENAZWANE

Mechanizm łączy nienazwanych można z powodzeniem wykorzystać do komunikacji dwukierunkowej, między procesem nadrzędnym a potomnym. Kolejny przykład pokazuje tego rodzaju wariant komunikacji międzyprocesowej.

Złóżmy, że proces nadrzędny prześle do procesu potomnego pewną wartość x oczekując na wykonanie na niej pewnej operacji f(x)=y a proces potomny zwróci wynik tej operacji, czyli y, do procesu nadrzędnego.

```
//bidir.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main( void )
{
    int pid,status;
    int one[2],two[2];
    double x=1.0,y=1.0;

    printf( "\t[%d] nadrzędny, start\n\n",(int)getpid() );
    // Oczywiście, w przypadku komunikacji dwukierunkowej,
    // konieczne są dwa łącza pipe
    if( pipe( one )< 0 || pipe( two )<0 )
    { printf( "<!> błąd otwarcia łączy\n" ); exit( 1 ); }
```

ŁĄCZA NIENAZWANE

```
switch( pid=(int)fork() )
{
    // Gdyby coś poszło nie tak
    case -1:
        perror( "<!> błąd inicjacji potomka" );
        exit( 1 );
        break;
    // Teraz kod dla potomka
    case  0:
        // Powitanie (jest to oczywiście czysta diagnostyka)
        printf( "\t[%d] potomek, start\n\n",(int)getpid() );
        // Zamykamy niepotrzebne deskryptory,
        // odpowiednio do kierunku przesyłu
        close( one[1] ); close( two[0] );
        // Nasłuchujemy tego co nadrzędny ma nam do powiedzenia
        read( one[0],(void*)&x,sizeof( double ) );
        printf( "\t[%d] otrzymał x=%f\n",(int)getpid(),x );
        // Wykonujemy właściwe operacje, na rzecz nadrzędnego
        y = x*M_PI;
        printf( "\t[%d] wykonał f(x)=y,wysyła y=%f\n",(int)getpid(),y );
        // no i w końcu wysyłamy wynik końcowy do nadrzędnego
        write( two[1],(void*)&y,sizeof( double ) );
        printf( "\t[%d] potomek, stop\n\n",(int)getpid() );
        exit( 0 );
        break;
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

ŁĄCZA NIENAZWANE

A teraz kod dla procesu nadrzędnego

```
default:
    // Oczywiście zamykamy to co nie jest nam potrzebne
    close( one[0] ); close( two[1] );
    // Wysyłamy dane do potomka
    printf( "\t[%d] wysyła do potomka [%d]x=%f\n\n",(int)getpid(),pid,x );
    write( one[1],(void*)&x,sizeof( double ) );
    // no i czekamy na wynik do potomka
    read( two[0],(void*)&y,sizeof( double ) );
    wait( &status );
    printf( "\t[%d] kod powrotu potomka [%d]\n",pid,status );
    //informacja diagnostyczna
    printf( "\t[%d] otrzymał y=%f\n",(int)getpid(),y );
    printf( "\n\t[%d] nadrzędny, stop\n",(int)getpid() );
}
//Proces nadrzędny kończy ostatecznie działanie
return 0;
}
```

KACZA NIENAZWANE

Po skompilowaniu źródeł

```
$ gcc bidir.c -o bidir
```

efekt wykonanie programu przedstawia się następująco

```
$ bidir
[7810] nadrzędny, start

[7811] potomek, start

[7810] wysyła do potomka [7811] x=1.000000

[7811] otrzymał x=1.000000
[7811] wykonał f(x)=y, wysyła y=3.141593
[7811] potomek, stop

[7811] kod powrotu z potomka [0]
[7810] otrzymał y=3.141593

[7810] nadrzędny, stop
$
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)