Łącza komunikacyjne nazwane

Prócz ewidentnych zalet i korzyści wynikających z użycia w komunikacji międzyprocesowej łączy nienazwanych, mimo wszystko nie zawsze warto (a nawet można) je stosować. Zasadniczym utrudnieniem w przypadku łączy nienazwanych jest kwestia współużytkowania takiego łącza przez procesy niespokrewnione.

W takiej sytuacji zwykle lepszym wyborem będą łącza w postaci potoku nazwanego (**named pipe**).

W odróżnieniu od nienazwanych, łącze nazwane (**named pipe**):

- jest identyfikowane przez nazwę i może z niego korzystać dowolny proces (także niespokrewniony), o ile ma odpowiednie uprawnienia;
- posiada organizację **FIFO**, czyli **First In First Out** (stąd i ich skrótowa nazwa);
- posiada dowiązanie w systemie plików (jako plik specjalny urządzenia), aż do momentu jawnego usunięcia;
- mimo iż zachowuje cechy pliku – jak wyjaśnia to dokumentacja systemowa LINUX:

... is a window into the kernel memory, that "looks" like a file ...

Obliczenia Równoległe i Systemy Rozproszone
Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Named pipe, podobnie jak i łącza nienazwane, mogą być tworzone w dwojaki sposób, a mianowicie z:

- systemowego interface'u użytkownika;
- procesu (czy wątku), wywołaniem odpowiedniej funkcji API systemowego.

Łącza komunikacyjne nazwane

Z poziomu interface systemowego łącza nazwane FIFO, tworzone są komendą

$ mkfifo -m mode name

mode maska praw dostępu, czyli symbolicznie dla **u** (user), **g** (group), **o** (other), **a** (all)
dodaje (**+**), ujmuje (**–**) od istniejących lub ustawia (**=**) prawo **r** (read), **w** (write), **x** (execute)
name nazwa pliku specjalnego FIFO, ewentualnie wraz ze ścieżką

Usunięcie łącza nazwanego odbywa się w identyczny sposób jak każdego pliku dyskowego, a więc

$ rm name

Utwórzmy w takim razie przykładowe łącze FIFO

$ mkfifo a=rw /tmp/km-fifo

Jeżeli wykonamy teraz

```
$ ls -l /tmp/km-*
prw-r--r-- 1 kmirota users 0 maj 22 12:09 /tmp/km-fifo
```

Zwróćmy uwagę na opis dowiązania, zgodnie ze specyfikacją dla **ls**

-	**Regular file**	**l**	**Symbolic link**
b	**Block special file**	**n**	**Network file**
c	**Character special file**	**p**	**FIFO**
d	**Directory**	**s**	**Socket**

Skoro **p** więc **FIFO pipe**.

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Takie cechy *pliku* **km-fifo** potwierdza także, w szczegółach, komenda **stat**

```
$ stat /tmp/km-fifo
File: `/tmp/km-fifo`
Size: 0                 Blocks: 0          IO Block: 4096   potok
Device: 806h/2054d      Inode: 591930      Links: 1
Access: (0644/prw-r--r--) Uid:(1000/kmirota) Gid:(100/users)
Access: 6003-05-22 12:35:52.000000000 +0200
Modify: 6003-05-22 12:35:51.000000000 +0200
Change: 6003-05-22 12:35:51.000000000 +0200
```

Otwórzmy teraz dwie sesje terminala, w oknie pierwszego pierwszego wpisujemy

```
$ cat < /tmp/km-fifo
```

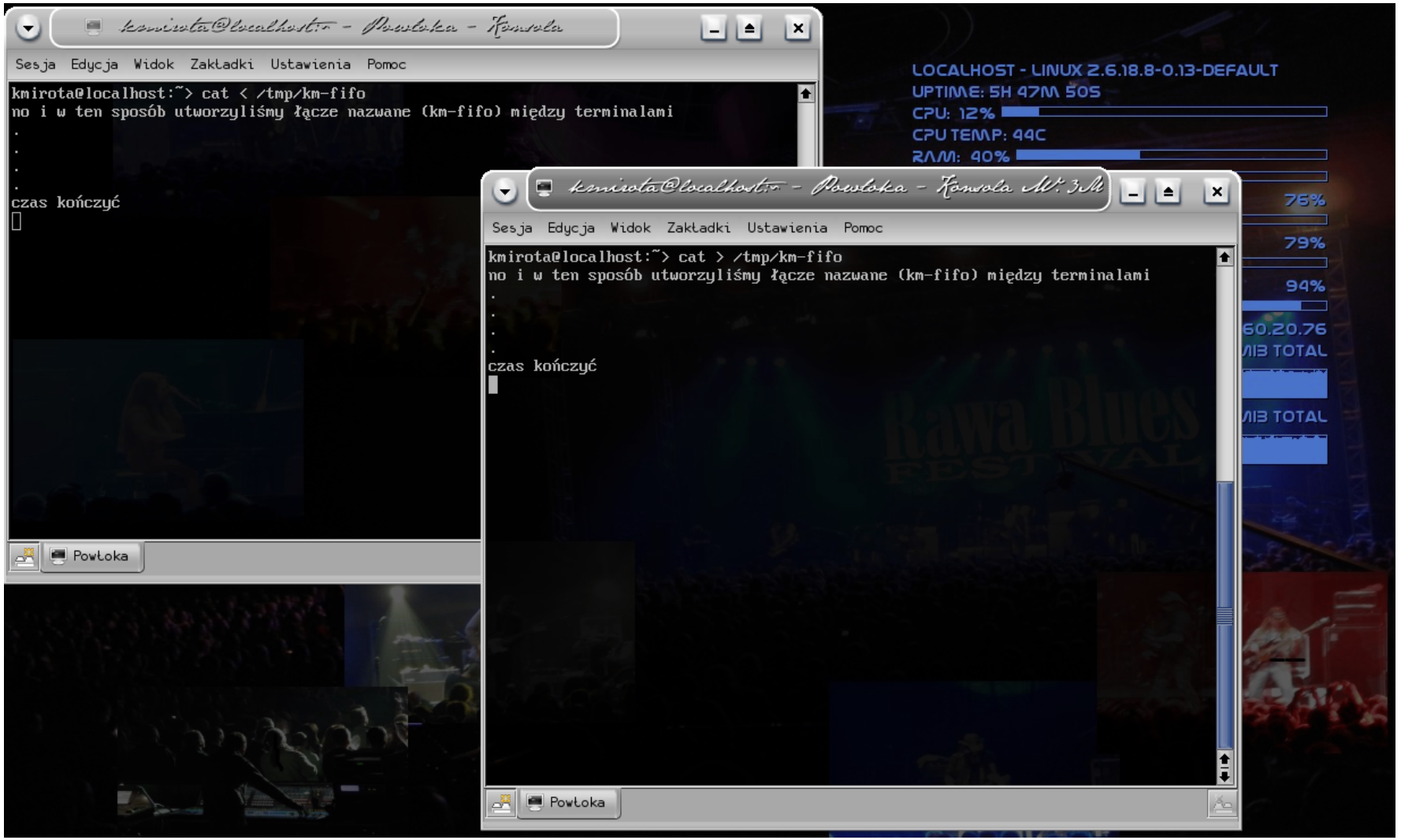
czyli przekierujemy zawartość **km-fifo** na wejście komendy systemowej **cat**. Ta, jak wiadomo, wyświetla na swoim wyjściu (czyli tutaj w oknie terminala pierwszego), to co otrzyma na wejściu. Teraz przełączamy się na drugie okno terminala, i wykonujemy

```
$ cat > /tmp/km-fifo
```

czyli w przeciwnym kierunku, zatem wejście drugiego terminala zostało przekierowane na plik **km-fifo**. Efekt będzie taki, że cokolwiek wprowadzimy w oknie terminala drugiego, natychmiast zobaczymy w oknie pierwszego. Zauważmy że *plik* ten będzie miał przez cały czas rozmiar zerowy.

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane



Całość można zakończyć przesyłając do **`cat`** kod **`EOF`** (End Of File), czyli z klawiatury **`<Ctrl>-d`**.

Obliczenia Równoległe i Systemy Rozproszone
Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Jeżeli na pierwszym terminalu ponownie wykonamy przekierowanie

```
$ cat < /tmp/km-fifo
```

zaś na drugim przekierujemy do pliku **km-fifo** wykonanie jakiejkolwiek komendy, przykładowo

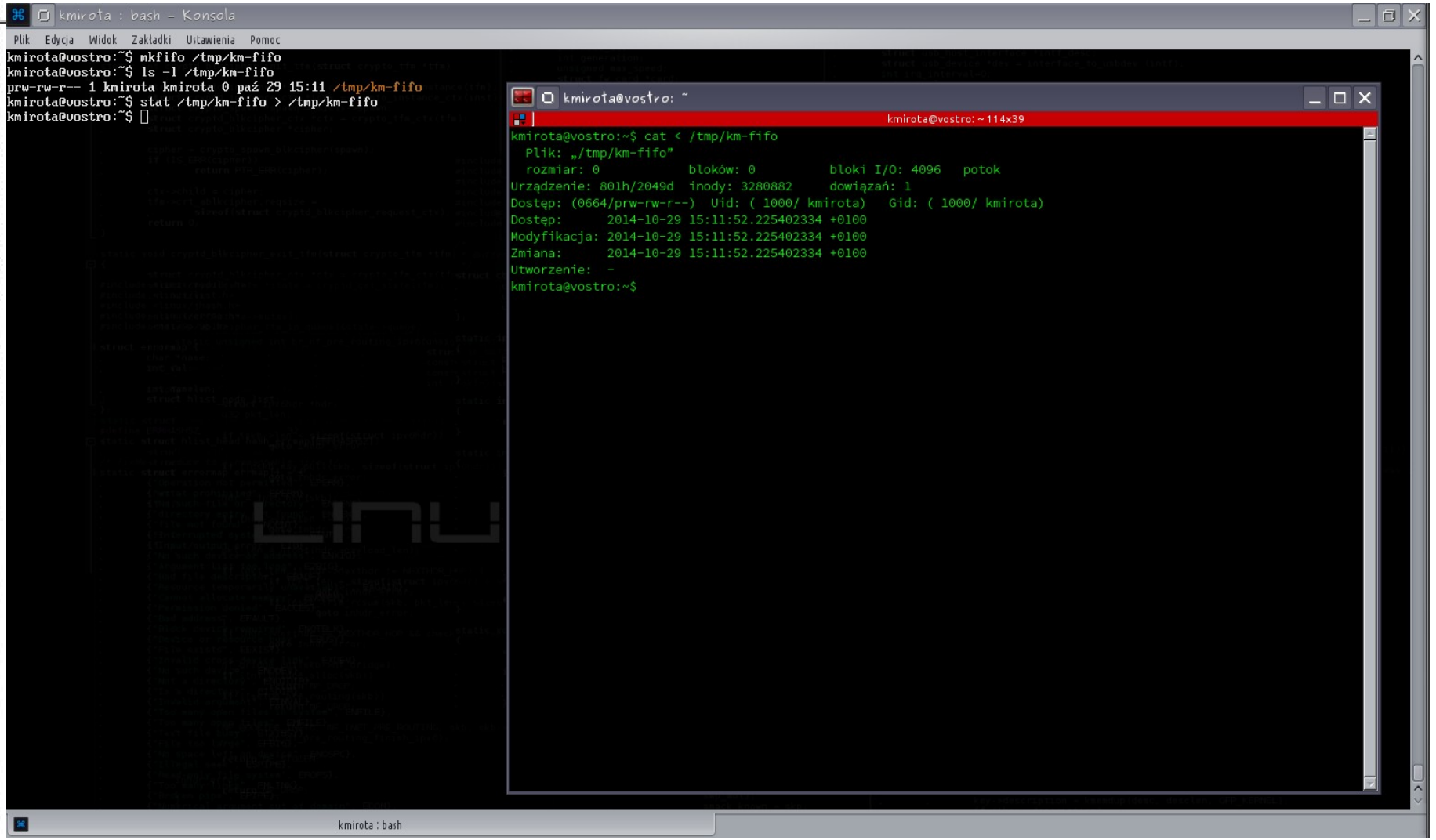
```
$ stat /tmp/km-fifo >/tmp/km-fifo
```

to na pierwszym zobaczymy oczywiście

```
File: `/tmp/km-fifo'
Size: 0                 Blocks: 0     IO Block: 4096   potok
Device: 806h/2054d      Inode: 591930 Links: 1
Access: (0644/prw-r--r--) Uid: (1000/kmirota)   Gid:(100/users)
Access: 6003-05-22 13:53:13.000000000 +0200
Modify: 6003-05-22 13:53:13.000000000 +0200
Change: 6003-05-22 13:53:13.000000000 +0200
$
```

i w tym momencie potok zostanie automatycznie zamknięty, bowiem strumień przepływających danych uległ zakończeniu.

Łącza komunikacyjne nazwane



Zauważmy że po zakończeniu sesji *plik* pozostaje w katalogu, a rozmiar jest zerowy. Ponieważ łącze usuwamy jak plik, więc

```
$ rm /tmp/km-fifo
```

Łącza komunikacyjne nazwane

W systemach rodziny POSIX, łącza nazwane tworzone są wywołaniami funkcji **`mkfifo()`**.

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>

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

`const char *pathname` nazwa tworzonego pliku (tu: łącza)

`mode_t mode` maska uprawnień odnośnie odczytu i zapisu

Return

`0` jeżeli zakończone sukcesem

Errors

`-1`

Usuwanie łączy nazwanych odbywa się przy pomocy funkcji **`unlink()`**.

Synopsis

```
#include <unistd.h>
int unlink( const char *pathname );
```

`const char *pathname` nazwa usuwanego pliku (tu: łącza)

Return

`0` jeżeli zakończone sukcesem

Errors

`-1`

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Maska uprawnień może być zadana z wykorzystaniem sumy bitowej predefiniowanych stałych w pliku nagłówkowym **sys/stat.h**:

S_IRUSR S_IWUSR S_IXUSR dla właściciela
S_IRGRP S_IWGRP S_IXGRP dla grupy, do której należy właściciel
S_IROTH S_IWOTH S_IXOTH dla pozostałych

lub też korzystając z stałej **DEFFILEMODE**, dla wszystkich łącznie

```
#define DEFFILEMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
```

ustawiając dla wszystkich uprawnienia do odczyty i zapisu (zwróćmy uwagę, że na efektywne uprawnienia mają wpływ stosowane ustalenia odnośnie katalogu nadrzędnego).

Ponieważ łącze nazwane posiada dowiązanie do struktury plików operacje na nim przeprowadza się w zwyczajowy sposób, tak jak i w przypadku ogółu, a więc:

- utworzone łącze trzeba otworzyć, przed użyciem, uzyskując w ten sposób jego deskryptor albo wskazanie do obiektu typu **FILE**;
- wykonujemy odczyt lub zapis, odpowiednio do potrzeb;
- po wykorzystaniu łącza proces powinien zamknąć je, od swojej strony.

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Niskopoziomowe operacje otwarcia i zamknięcia realizujemy przy pomocy **`open()`** i **`close()`**.

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open( const char *pathname, int flags );
```

`const char *pathname` nazwa otwieranego pliku (tu: łącza)

`int flags` maska określająca sposób korzystania:

`O_RDONLY O_WRONLY O_RDWR` tylko odczyt, tylko zapis, odczyt lub zapis

`O_NONBLOCK` otwarcie w trybie nieblokującym

Return

deskryptor pliku

Errors

`-1`

Synopsis

```
#include <unistd.h>
int close( int fd );
```

`int fd` deskryptor pliku (tu: łącza nazwanego)

Return

`0` zakończona sukcesem

Errors

`-1`

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Przydzielając deskryptor łączu za pośrednictwem funkcji **`open()`** trzeba pamiętać, że jej wywołanie mają zasadniczo charakter blokujący, tzn. wywołanie
`fd=open( pathname,O_RDONLY );` będzie czekać aż jakiś proces nie otworzy kolejki do zapisu
`fd=open( pathname,O_WRONLY );` będzie czekać aż jakiś proces nie otworzy kolejki do odczytu

Jeżeli jednak otwarcie nastąpi z użyciem maski **`O_NONBLOCK`** wówczas
`fd=open( pathname,O_RDONLY|O_NONBLOCK );` zwróci natychmiast sterowanie
`fd=open( pathname,O_WRONLY|O_NONBLOCK );` także zwróci natychmiast sterowanie, jeżeli jednak nie będzie żadnego procesu, który wykona otwarcie do odczytu, to zamiast deskryptora otrzymamy
`-1` (oraz **`errno = ENXIO`**)

Dysponując deskryptorem, do odczytu i zapisu, używamy **`read()`** i **`write()`**. - podobnie jak w przypadku łączy nienazwanych.

Synopsis

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

`int fd` deskryptor pliku
`void* buffer` bufor do odczytu lub zapisu
`size_t bytes` ilość bajtów do przesłania

Return

ilość bajtów przesłanych

Errors

`-1`

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

O ile zasadniczym przeznaczeniem łączy nazwanych jest wymiana informacji między procesami niespokrewnionymi, to może równie dobrze odbywać się w obrębie pojedynczego procesu.

Ilustruje to kolejny przykład.

```
#include <unistd.h>
#include <sys/stat.h>
#include <errno.h>   // kody błędów errno
#include <string.h>  // opisy błędów errno
#include <fcntl.h>   // definicje dla open()
#include <linux/limits.h> // definicja PIPE_BUF
#include <stdio.h>

int main( void )
{
    char pipe[]="xyz"; // to będzie nasze łącze nazwane
    char message[PIPE_BUF]="POZDROWIENIE (OD SAMEGO SIEBIE)";
    int in,out; // deskryptory dla wejścia i wyjścia

    // próbujemy utworzyć łącze nazwane
    if( mkfifo( pipe,S_IRUSR|S_IWUSR )==0 || errno == EEXIST )
    {
        printf( "errno=%d ...%s... mkfifo() \n",errno,strerror(errno) );
```

Łącza komunikacyjne nazwane

Zwróćmy uwagę na instrukcję warunkową

```
if( mkfifo( pipe,S_IRUSR|S_IWUSR )==0 || errno == EEXIST )
{

}
```

Wprowadzenie warunku w postaci alternatywy

```
mkfifo( pipe,S_IRUSR|S_IWUSR )==0 || errno == EEXIST
```

wynika z faktu, że nie można utworzyć łącza o nazwie już istniejącego (bowiem ma dowiązanie w systemie plików). Jeżeli pojawi się błąd

```
EEXIST
```

będzie użyte łącze już istniejące.

Próbujemy otworzyć łącze do zapisu i otwarcia.

```
if( (in = open( pipe, O_RDONLY|O_NONBLOCK) ) < 0 )
{ printf( "errno=%d ...%s... mkfifo() \n",errno,strerror(errno) ); }
else
{
    if( (out = open( pipe, O_WRONLY)) < 0 )
    { printf("errno=%d ...%s... mkfifo() \n",errno,strerror(errno)); }
```


Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

W końcu, czas skorzystać z łącza

```
            else
            {
                write( out,message,PIPE_BUF );
                read(  in,message,PIPE_BUF );
                close( out );
                printf( "%s\n",message );
            }
            close( in );
        }
        // usunięcie łącza (niekoniecznie musi tak być)
        if(unlink( pipe )==-1 )
        {  perror( "unable to remove named pipe" ); }
    }
    else
    {
        printf( "errno=%d ...%s... %s\n",errno,strerror(errno),
        "Unable to create named pipe" );
    }

    return 0;
}
```


Kryspin Mirota (kryspin.mirota@gmail.com)

ŁĄCZA KOMUNIKACYJNE NAZWANE

Kompilacja

```
kmirota@vostro:~$ gcc -Wall fifo.c -o fifo
```

a efekt użycia

```
kmirota@vostro:~$ ./fifo
errno=0 ...Success... mkfifo()
POZDROWIENIE (OD SAMEGO SIEBIE)
```

Można dodać, że fragment kodu dotyczący przesłania komunikatu

```
else
{
    write( out,message,PIPE_BUF );
    read(  in,message,PIPE_BUF );
    close( out );
    printf( "%s\n",message );
}
```

Można byłoby zapisać także w nieco innej formie

```
else
{
    write( out,message,PIPE_BUF );
    read(  in,message,PIPE_BUF );
    close( out );
    write(STDOUT_FILENO,message,n); //zamiast printf("%s\n",message);
}
```

tyle że wcześniej trzeba byłoby pamiętać o

```
    n = read(  in,message,PIPE_BUF );
```

Łącza komunikacyjne nazwane

Na łączu nazwanym możemy także operować w sposób bezpośredni za pomocą wysokopoziomowych funkcji ISO C standard I/O.

W pierwszej kolejności należy oczywiście otworzyć łącze wywołaniem **`fopen()`**.

Synopsis

```
#include <stdio.h>
FILE *fopen( const char *pathname,const char *mode );
```

`const char *pathname` nazwa otwieranego pliku (tu: łącza)

`char *mode` tryb otwarcia pliku: **`r (r+)`** odczyt (i zapis),
`w (w+)` zapis (i odczyt), **`a (a+)`** rozszerzenie (i odczyt)

Return

wskazanie do strumienia plikowego **`FILE`**

Errors

`NULL`

W końcu zaś zamknąć przy pomocy **`fclose()`**.

Synopsis

```
#include <stdio.h>
int fclose( FILE *stream );
```

`FILE *stream` wskazanie do strumienia, do zamknięcie

Return

`0` jeżeli sukces

Errors

`EOF`

Łącza komunikacyjne nazwane

Celem odczytu i zapis z i do strumienia plikowego stosuje się głównie funkcje formatowanego wejścia/wyjścia **`fscanf()`** i **`fprintf()`**.

Synopsis

```
#include <stdio.h>
int fscanf( FILE *stream,const char *format, ...  );
int fprintf( FILE *stream,const char *format, ... );
```

`FILE *stream` wskazanie do strumienia otwartego do odczytu

`const char *format` łańcuch formatujący zawierający wzorce konwersji, w szczególności: **`"%c"`** znak,**`"%s"`** łańcuch, **`"%d"`** liczba całkowita, **`"%f"`** liczba rzeczywista

`...` lista zmiennych

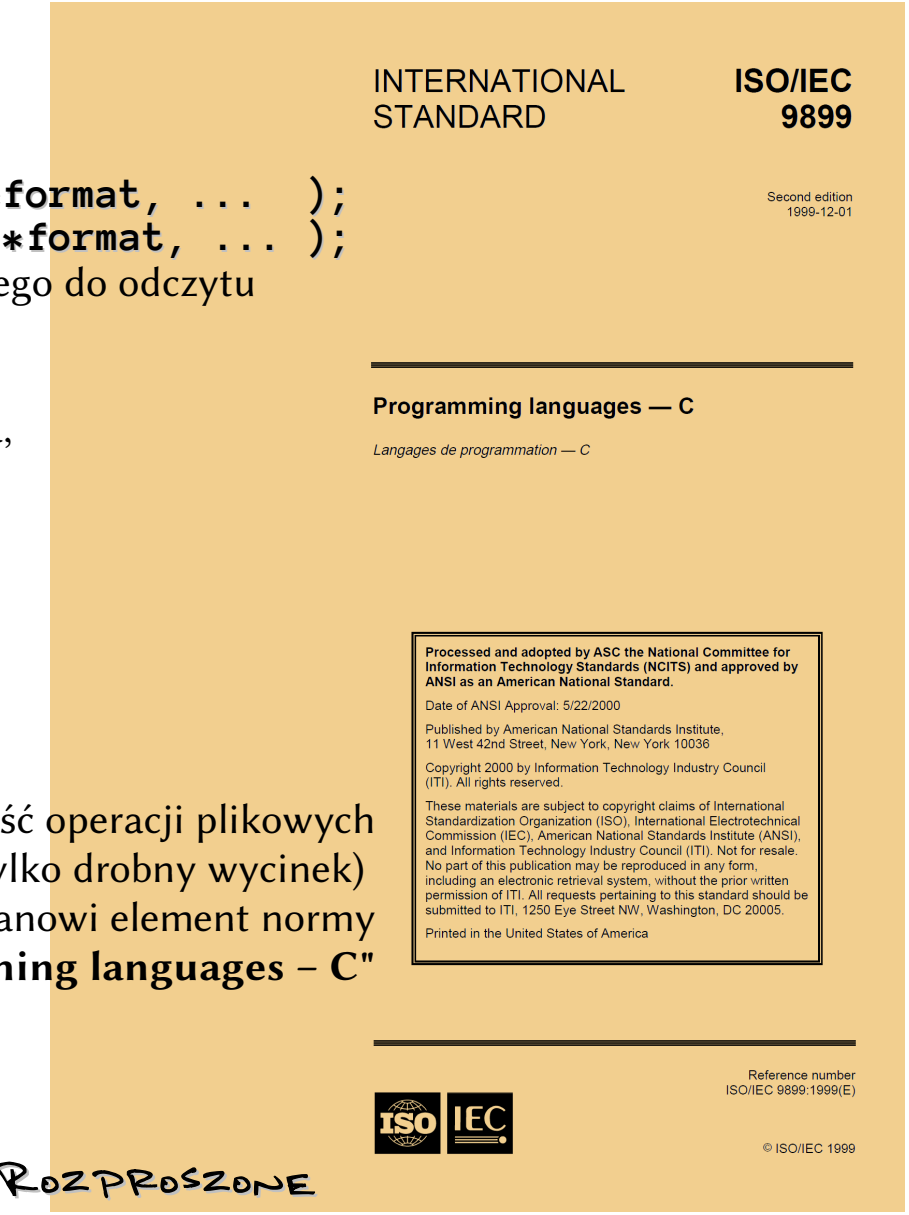
Return

ilość zmiennych wczytanych skutecznie, wg listy argumentów

Errors

`EOF`

Całość operacji plikowych
(z czego tu przedstawiono tylko drobny wycinek)
stanowi element normy
ISO/IEC 9899:1999 "Programming languages – C"



INTERNATIONAL STANDARD **ISO/IEC 9899**

Second edition
1999-12-01

Programming languages — C

Langages de programmation — C

Processed and adopted by ASC the National Committee for Information Technology Standards (NCITS) and approved by ANSI as an American National Standard.

Date of ANSI Approval: 5/22/2000

Published by American National Standards Institute,
11 West 42nd Street, New York, New York 10036

Copyright 2000 by Information Technology Industry Council (ITI). All rights reserved.

These materials are subject to copyright claims of International Standardization Organization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and Information Technology Industry Council (ITI). Not for resale. No part of this publication may be reproduced in any form, including an electronic retrieval system, without the prior written permission of ITI. All requests pertaining to this standard should be submitted to ITI, 1250 Eye Street NW, Washington, DC 20005.

Printed in the United States of America

Reference number
ISO/IEC 9899:1999(E)

© ISO/IEC 1999

Obliczenia Równoległe i Systemy Rozproszone
Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Wejście i wyjście plikowe, z punktu widzenie programisty, przedstawia się zwykle dość korzystnie i atrakcyjnie, jednak może przysporzyć także problemy. Jedną z przyczyn mogą być stosowanych tu mechanizmów buforowania. W konsekwencji mogą pojawić się niezgodności tego co zawarte jest w skojarzonym buforze plikowym a wartościami zmiennych, póki strumień nie zostanie zamknięty.

Przykładem tego rodzaju sytuacji jest poniższy kod.

```
#include <stdio.h>
int main( void )
{
    printf( "1 sek " ); sleep( 1 );
    printf( "2 sek " ); sleep( 1 );
    printf( "3 sek " ); sleep( 1 );
    printf( "4 sek " ); sleep( 1 );
    printf( "5 sek " ); sleep( 1 );

    printf( "i koniec\n" );

    return 0;
}
```


Kryspin Mirota (kryspin.mirota@gmail.com)

W następstwie wykonania programu, nie zobaczymy wcale serii komunikatów (choć zależy to od użytego kompilatora), pojawiających się co 1 sekundę, ale wszystkie równocześnie w momencie kiedy napotkany będzie znak końca linii **'\n'**.

```
1 sek 2 sek 3 sek 4 sek 5 sek i koniec
```

Aby osiągnąć zamierzony efekt, wyświetlanego komunikatu co sekundę należałoby użyć funkcji **fflush()** opróżniającej bufor plikowy, tutaj standardowego wyjścia **stdout**.

Synopsis

```
#include <stdio.h>
int fflush( FILE *stream );
```

FILE *steram bufor plikowy do opróżnienia (uprzednio otwarty)

Return

0 jeżeli sukces

Errors

EOF

Zatem w przypadku rozpatrywanego kodu konieczne jest jego uzupełnienie o wywołania

```
fflush( stdout );
```

w sposób ja na załączonym dalej listingu.

Łącza komunikacyjne nazwane

```
#include <stdio.h>

int main( void )
{
    printf( "1. " ); fflush( stdout); sleep( 1 );
    printf( "2. " ); fflush( stdout); sleep( 1 );
    printf( "3. " ); fflush( stdout); sleep( 1 );
    printf( "4. " ); fflush( stdout); sleep( 1 );
    printf( "5. " ); fflush( stdout); sleep( 1 );

    printf( "i koniec\n" );

    return 0;
}
```

O samej funkcji warto pamiętać ponieważ problem buforów plikowych w odniesieniu do łączy komunikacyjnych bywa nad wyraz uciążliwy, stając się przyczyną wielu zaskakujących wyników i frustracji.

Na koniec należy nadmienić że o ile użycie

```
fflush( stdout )
```

jest zgodne z normą ISO, to w odniesieniu do strumienia wejściowego

```
fflush( stdin )
```

już nie jest.

Łącza komunikacyjne nazwane

Problem buforowania po stronie odczytu można także rozwiązać używając celem wczytywania funkcji "nie zaśmiecającej" tak bufor plikowy jak **`scanf()`**. Mogłaby to być – w tym przypadku – **`fgets()`**, która wczytuje łańcuch tekstowy, a dopiero z niego za pomocą **`sscanf()`** wczytujemy potrzebne wartości zmiennych.

Synopsis

```
#include <stdio.h>
char *fgets(char *buffer, int size, FILE *stream);
```

`char *buffer` bufor odczytu
`int size` ilość znaków do wczytania, pomniejszona o 1, na ostatniej pozycji dodawane jest **`'\0'`** (funkcja kończy czytanie jeżeli napotka **`EOF`** lub **`'\n'`**)
`FILE *stream` otwarty do odczytu bufor plikowy

Return

`char *buffer` jeżeli sukces, to wskazanie do wyniku

Errors

`NULL`

Deklaracja i użycie funkcji

```
int sscanf( const char *buffer,const char *format, ... );
```

jest właściwie identyczne z **`fscanf()`**, w tym że odczyt nie następuje ze strumienia plikowego ale łańcucha

`const char *buffer` bufor w postaci łańcucha tekstowego, z którego zawartość podlega konwersji, zgodnie z łańcuchem format

Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Przygotujemy teraz dwie aplikacje, które poprzez łącze nazwane **pipe** będą wzajemnie się komunikować.

Pierwsza stanowić będzie serwer nasłuchujący danych napływających łączem, a w przypadku odebrania wyświetli komunikat.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>

int main( void )
{
     time_t stamp;
     pid_t pid;
     int fd,run;
     char cmd;

     printf("\n[%d]*S*E*R*W*E*R*[%d]\n\n",(int)getpid(),(int)getpid());

     // na początek próbujemy otworzyć łącze do odczytu
     if( (fd=open( "pipe",O_RDONLY )) >0 ){ run=1; }
     else{ printf( "!.!..nie znaleziono łącza..!.!\n\n" ); run=0; }
```

ŁĄCZA KOMUNIKACYJNE NAZWANE

```
    // w przypadku kiedy udało się uzyskać deskryptor
    while( run )
    {
        // odczytujemy, od kogo pochodzi wiadomość
        read( fd,&pid,sizeof( pid_t ) );
        // następie czytamy komendę
        read( fd,&cmd,sizeof( char ) );
        // wyświetlamy informację, od kogo, co i kiedy otrzymano
        stamp = time( NULL );
        printf( "[%d]\t|%c|-> %s",(int)pid,cmd,ctime( &stamp ) );
        // jeżeli odebrano komendę Q(uit), to kończymy
        if( cmd=='Q' ){ run=0; close( fd ); }
    }

    return 0;
}
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

Łącza komunikacyjne nazwane

Teraz kod klienta, który będzie komunikował się z naszym serwerem przez łącze **pipe**.

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

int main( void )
{
    int fd,run;
    pid_t pid;
    char cmd;
    char buffer[256];

    // wyświetlamy komunikat diagnostyczny
    pid = getpid();
    printf( "\n[%d]*K*L*I*E*N*T*[%d]\n\n",(int)pid,(int)pid );

    // podobnie jak wcześniej sprawdzamy dostępność łącza
    if( (fd=open( "pipe",O_WRONLY )) >0 ){ run=1; }
    else{ printf( "!.!..nie znaleziono łącza..!.!\n\n" ); run=0; }
```

ŁĄCZA KOMUNIKACYJNE NAZWANE

```
    // jeżeli udało się uzyskać deskryptor łącza
    while( run )
    {
        // pobieramy komendę od użytkownika
        printf( "\t?...\t" );
        fgets( buffer,256,stdin );
        sscanf( buffer,"%c",&cmd );
        cmd = toupper( cmd );

        // piszemy do łącza
        write( fd,&pid,sizeof( pid_t ) );
        write( fd,&cmd,sizeof( char ) );

        // jeżeli użytkownik podał Q(uit), to kończymy proces klienta
        if( cmd=='Q' ){ run=0; close( fd ); }
    }

    return 0;
}
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

Łącza komunikacyjne nazwane

Zanim uruchomimy procesy serwer i klienckie, najpierw musimy utworzyć łącze.

Zatem

```
$ mkfifo pipe
```

Sprawdźmy jeszcze wynik komendy

```
$ ls -l
razem 32
-rwxr-xr-x 1 kmirota users 11667 maj 23 10:36 klient
-rw-r--r-- 1 kmirota users   697 maj 23 10:36 klient.c
prw-r--r-- 1 kmirota users     0 maj 23 10:37 pipe
-rwxr-xr-x 1 kmirota users 11576 maj 23 10:36 serwer
-rw-r--r-- 1 kmirota users   682 maj 23 10:36 serwer.c
```

a więc łącze pipe zostało utworzone.

Otwieramy teraz dwa terminale w pierwszym uruchamiamy proces serwera a w drugim (i ewentualnie kolejnych) procesy klienckie.

```
$ ./klient

[8053]*K*L*I*E*N*T*[8053]

        ?...    ?
        ?...    ?
        ?...    ?
        ?...    q
```

```
$ ./serwer

[8052] * S * E * R * W * E * R * [8052]

[8053]  |?|-> Fri Oct 23 11:07:48 2003
[8053]  |?|-> Fri Oct 23 11:07:50 2003
[8053]  |?|-> Fri Oct 23 11:07:51 2003
[8053]  |Q|-> Fri Oct 23 11:07:54 2003
```


Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Ponownie wykorzystując mechanizm łącza nazwanego przygotujemy serwer, który pozostając na końcu łącza zamknie je i będzie odsyłał zwrotnie cokolwiek dostanie.

Kod źródłowy serwera przedstawia się jak na załączonym listingu.

```
#include <stdio.h>
#include <limits.h>
int main( void )
{
    FILE *stream;
    char buffer[LINE_MAX];
    int run;

    if( (stream = fopen( "channel","r+" ) ) ){ run=1; }
    else{ run=0; perror( "!.!..błąd otwarcia łącza..!.!" ); }

    while( run )
    {
        if( fgets( buffer,256,stream ) )
        { fprintf( stream,"%s",buffer ); fflush( stream ); }
    }

    return 0;
}
```

Zwróćmy uwagą, że celem wyświetlenia informacji o ewentualnym błędzie otwarcia kanału, użyliśmy funkcji **perror()**, gdyż tylko ona gwarantuje że nawet w przypadku procesu pracującego w tle zobaczymy komunikat o błędzie.

ŁĄCZA KOMUNIKACYJNE NAZWANE

Początkowy fragment kodu klienta niewiele tylko różni się od przedstawionego wcześniej.

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <ctype.h>

int main( void )
{
    FILE *stream;
    char buffer[LINE_MAX];
    int run;

    int empty( char* );

    if( (stream = fopen( "channel","r+" ) ) ){ run=1; }
    else{ run=0; perror( "!.!..błąd otwarcia łącza..!.!" ); }
```


KRYSPIN MIROTA (KRYSPIN.MIROTA@GMAIL.COM)

Łącza komunikacyjne nazwane

W dalszym fragmencie umieszczamy pętle czytającą i piszącą, do momentu kiedy użytkownik wprowadzi linię pustą.

```
    while( run )
    {
        bzero( (void*)buffer,LINE_MAX );
        fgets( buffer,LINE_MAX,stdin );
        if( !empty( buffer ) )
        { fprintf( stream,"%s",buffer ); fflush( stream ); }
        else{ fclose( stream ); break; }
        fgets( buffer,LINE_MAX,stream );
        if( !empty( buffer ) )
        { fprintf( stdout,"%s",buffer ); fflush( stream ); }
    }

    return 0;
}
```

Pozostaje jeszcze dodać funkcję która będzie sprawdzać czy nie pojawił się łańcuch pusty (w naszym rozumieniu łańcuch nie zawierający ani jednej litery czy też cyfry).

```
int empty( char * string )
{
    while( *string ){ if( isalnum(*string++) ){ return 0; } }
    return 1;
}
```

Łącza komunikacyjne nazwane

Oczywiście na początek należy utworzyć właściwe łącze nazwane

```
$ mkfifo channel
```

i sprawdźmy

```
$ stat channel
  File: `channel'
  Size: 0                 Blocks: 0          IO Block: 4096   potok
Device: 807h/2055d        Inode: 4162321     Links: 1
Access: (0644/prw-r--r--) Uid:(1000/kmirota) Gid: (100/users)
Access: 2006-12-23 06:29:21.000000000 +0200
Modify: 2006-12-23 06:29:21.000000000 +0200
Change: 2006-12-23 06:29:21.000000000 +0200
```

Następnie uruchamiamy z bieżącego terminala proces serwera, w tle (zwróćmy uwagę na znak **'&'** na końcu linii komendy, który wywołuje właśnie takie uaktywnienie procesu przez system).

```
$ ./serwer &
[1] 8603
```

gdyby teraz sprawdzić listę aktywnych procesów

```
$ ps
PID TTY          TIME CMD
4311 pts/1    00:00:00 bash
8603 pts/1    00:00:00 server
8607 pts/1    00:00:00 ps
```

to jak proces o identyfikatorze **8603** pracuje nasz serwer. Jego działanie możemy w dowolnej chwili zakończyć, komendą (o ile nie będzie już potrzebny)

```
$ kill -SIGKILL 8603
```


Kryspin Mirota (kryspin.mirota@gmail.com)

Łącza komunikacyjne nazwane

Następnie z bieżącego terminala uruchamiamy klienta. Ponieważ serwer pracuje zamyka pętlę zwrotną, czyli odsyła to co otrzymał, więc cokolwiek wprowadzimy zostanie nam natychmiast odesłane przez serwer.

```
$ ./client
pierwszy
pierwszy
drugi
drugi
i jeszcze nieco dłuższy łańcuch, 1234567890
i jeszcze nieco dłuższy łańcuch, 1234567890

$
```

a więc, po wprowadzeniu linii pustej, klient zakończył.

Pozostaje nam jeszcze zamknąć serwer. Tak jak sugerowano to wcześniej wykorzystamy w tym celu sygnał **SIGKILL**, zatem z terminala wykonujemy

```
$ kill -SIGKILL 8603
```