

KOMUNIKACJA MIĘDZYPROCESOWA POTOKI

POTOK (PIPE)

Indeks do tablicy plików - ***deskryptor pliku***

Zarezerwowane deskryptory:

STDIN 0

STDOUT 1

STDERR 2

- Łączy nienazwane
 - wysokopoziomowe fcje - **popen**, **pclose**
(strumienie plikowe FILE)
 - niskopoziomowe fcje - **pipe**, **dup**, **dup2**
(deskryptory)
- Potoki nazwane (pliki FIFO)

Potok nienazwany

- metoda przekierowywania standardowego wyjścia jednego procesu na standardowe wejście innego
- jednokierunkowy
- szeregowy
- nie ma dowiązania w systemie plików (w przeciwieństwie do kolejek FIFO)
- opisywany przez parę deskryptorów
- po zamknięciu deskryptorów przestaje istnieć
- komunikacja między wątkami jednego procesu lub między procesami rodzicielskim a potomnym
- synchronizacja

Potok a plik

- Potok
 - ograniczony rozmiar (4kB – 8kB)
 - dostęp sekwencyjny (bez lseek)
 - odczyt niszczący
 - read – blokowane w oczekiwaniu na dane
 - write – blokowane, gdy nie ma miejsca (chyba że flaga O_NONBLOCK O_NDELAY)

Wysokopoziomowe funkcje

popen **pclose**

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char  
    *open_mode)
```

- uruchamia nowy proces (command);
- przekazuje (fwrite) (open_mode=w)
- lub odbiera od niego (fread)(open_mode=r) dane przez strumień FILE
- nowy proces korzysta z stdin i stdout

```
int pclose(FILE *stream_to_close)
```

czeka na zakończenie procesu uruchomionego przez popen

popen open_mode = w r

```
FILE* str=popen(„sort”,„w”);
fprintf(str,„tekst1\n”);
fprintf(str,„tekst2\n”);
fprintf(str,„tekst3\n”);
fprintf(str,„tekst4\n”);
fclose(str);
.
.
.
FILE* str=popen(„cat *.c”,„r”);
n=fread(bufor, sizeof(char), BUFSIZE, str);
while (n>0) {
printf(„.....”,bufor);
n=fread(bufor, sizeof(char), BUFSIZE, str);
}
fclose(str);
```

fwrite

popen

- umożliwia przesyłanie danych w małych porcjach szeregując procesy (zapis, buforowanie, odczyt)
- **popen** wywołuje powłokę i przekazuje jej polecenie jako argument
 - „+” - można stosować rozwinięcia parametryczne
 - „-” -kosztowna

potoki - funkcje niższego poziomu

pipe

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

zwraca:

0 - sukces

-1 - błąd

file_desc[2] - tablica deskryptorów:

file_desc[0] - odczyt,

file_desc[1] - zapis

(FIFO)

Dostęp do danych za pomocą fcji read, write (niskopoziomowe)

+fork – przekazywanie danych pomiędzy procesami
(dziedziczenie deskryptorów)


pot1

nr deskryptora do
odczytu – do bufora
-> arg. pot2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
    int prz_dane;
    int pot_pl[2];
    const char dane[]="abc";
    char bufor[BUFSIZ+1];
    int po_fork;
    memset(bufor,'\0',sizeof(bufor));
    if (pipe(pot_pl)==0)
    {
        po_fork=fork();
        if (po_fork ==-1)
            {fprintf(stderr,"blad fork");
            exit(EXIT_FAILURE);}
        if (po_fork == 0)
            {sprintf(bufor,"%d",pot_pl[0]);
            execl("pot2","pot2",bufor,NULL);
            exit(1);}
        else
            {prz_dane=write(pot_pl[1],dane,strlen(dane));
            printf("pid %d zapisał %d bajt.\n",getpid(),prz_dane);}
    }
    exit(0);
}
```

pot2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int prz_dane;
    int des_pl;
    char bufor[BUFSIZ+1];
    int po_fork;
    memset(bufor, '\0', sizeof(bufor));
    sscanf(argv[1], "%d", &des_pl);
    prz_dane = read(des_pl, bufor, BUFSIZ);
    printf("pid %d odczytał %d bajt...: %s\n",
        getpid(), prz_dane, bufor);
    exit(0);
}
```



pobranie nr deskryptora – des_pl

```
[ania@marcys potoki]$ ./pot1
pid 1519 zapisał 3 bajt..
pid 1520 odczytał 3 bajt...: abc
```

- read - zwykle blokuje proces w oczekiwaniu na dane; jeśli potok nie jest otwarty do zapisu (w przypadku fork w obu procesach) – zwraca 0 (uniknięcie zablokowania)

Łączenie procesów potokiem

duplikacja deskryptora

```
#include <unistd.h>
```

```
int dup( int file_desc);
```

```
int dup2(int file_desc1, int file_desc2);
```

Zwraca:

- nowy deskryptor (dup – najniższy dostępny numer)
- -1 - błąd

dup, dup2 - stary deskryptor nie jest zamykany,
można ich używać zamiennie

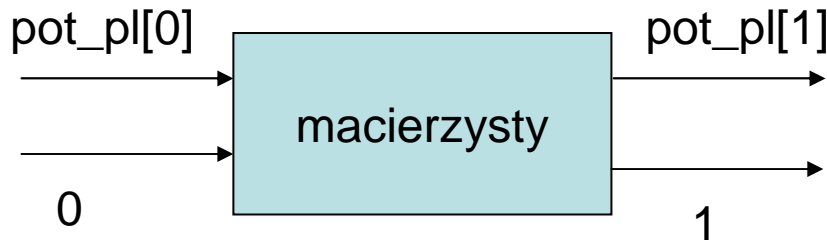
dup_1

```
int prz_dane;
int pot_pl[2];
const char dane[]="abc";
char bufor[BUFSIZ+1];
int po_fork;
if (pipe(pot_pl)==0)
{
    po_fork=fork();
    if (po_fork ==-1)
        {fprintf(stderr,"blad fork");
        exit(EXIT_FAILURE);}
    if (po_fork == 0)
        {close(0);
        dup(pot_pl[0]);
        close(pot_pl[0]);
        close(pot_pl[1]);
        execl("dup_2","dup_2",NULL);
        exit(1);}
    else
        {close(pot_pl[0]);
        prz_dane=write(pot_pl[1],dane,strlen(dane));
        close(pot_pl[1]);
        printf("pid %d zapisał %d bajt...:%s\n",getpid(),prz_dane,dane);}
}
exit(0);
```

dup_2

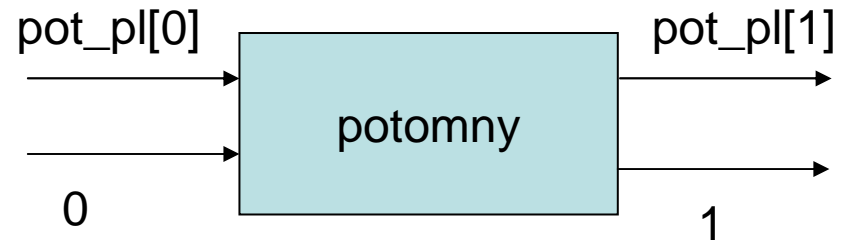
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int prz_dane;
    int des_pl;
    char bufor[BUFSIZ+1];
    memset(bufor, '\0', sizeof(bufor));
    gets(bufor);
    printf("pid %d odczytał : %s\n", getpid(), bufor);
    exit(0);
}
```

```
[ania@marcys potoki]$ ./dup
pid 1825 zapisał 3 bajt.:abc
pid 1826 odczytał : abc
```



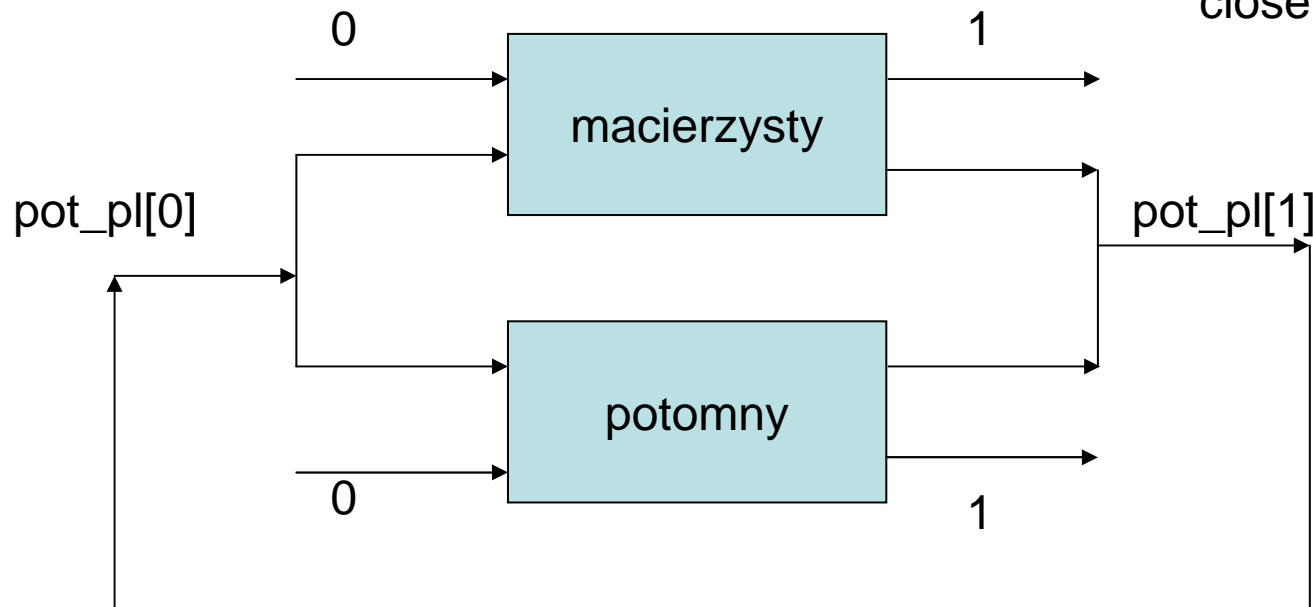
macierzysty

```
close(pot_pl[0]);
prz_dane=write(pot_pl[1],dane,strlen(dane));
close(pot_pl[1]);
```



potomny

```
close(0);
dup(pot_pl[0]);
close(pot_pl[0]);
close(pot_pl[1])
```



FIFO

łącza nienazwane – ograniczone do procesów spokrewnionych

Pliki FIFO – jest plikiem typu p, ma charakter potoku nienazwanego

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

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

```
int mkfifo(const char *filename, mode_t mode)
```

```
int mknod(const char *filename, mode_t mode | S_IFIFO,  
          (dev_t) 0)
```

mknod - można użyć do tworzenia wielu rodzajów plików, S_IFIFO w trybie dostępu oraz dev_t = 0 specyfikują powstanie FIFO

- FIFO można stosować we wszystkich poleceniach, które wymagają nazwy pliku
- przed zapisaniem lub odczytaniem czegoś do/z FIFO trzeba go otworzyć za pomocą funkcji **open**, której przekazuje się ścieżkę do FIFO
- do zamknięcia FIFO służy funkcja **close**
- ograniczenie - brak możliwości otworzenia go jednocześnie do zapisu i odczytu (**w trybie O_RDWR**)
- **FIFO służy zwykle do przesyłania danych w jednym kierunku**
- jeśli trzeba przesłać dane między programami w obu kierunkach można użyć pary FIFO lub jawnie zmienić kierunek przepływu danych poprzez zamknięcie i ponowne otwarcie FIFO

FIFO - otwieranie

Otwarcie zwykłego pliku a otwarcie FIFO

Istnieją cztery poprawne kombinacje znaczników:
O_RDONLY, O_WRONLY i O_NONBLOCK:

1. **open(const char *path, O_RDONLY);**
blokuje się dopóki inny proces nie otworzy potoku do zapisu
2. **open(const char *path, O_RDONLY | O_NONBLOCK);**
nie blokuje się
3. **open(const char *path, O_WRONLY);**
blokuje się dopóki inny proces nie otworzy potoku do odczytu
4. **open(const char *path, O_WRONLY | O_NONBLOCK);**
nie blokuje się , ale jeśli żaden proces nie otworzył FIFO do odczytu zwraca błąd (-1)

Najczęstsze zastosowanie nazwanych potoków:

proces czytający - O_RDONLY

proces piszący - O_WRONLY | O_NONBLOCK

- Czytający proces uruchamia się, czeka na powrót funkcji open, a kiedy inny program otworzy FIFO do zapisu oba programy kontynuują działanie
- Procesy synchronizują się przez wywołanie open

Zapis do FIFO, które nie może przyjąć wszystkich bajtów może skończyć się w dwojaki sposób:

- Spowodować błąd jeśli zażądano zapisu PIPE_BUF (limits.h - 4096 bajtów) lub mniejszej liczby bajtów, a dane nie mogą zostać przyjęte,
- Zapisać część danych, jeśli zażądano zapisu więcej niż PIPE_BUF bajtów, zwracając liczbę faktycznie zapisanych danych (może być równa 0)

System gwarantuje, że zapis PIPE_BUF lub mniejszej ilości bajtów do FIFO otwartego w trybie O_WRONLY (blokującego się), zapisze wszystkie bajty albo żadnego.

Jeśli kilka programów próbuje jednocześnie zapisać dane do FIFO istotne jest żeby bloki pochodzące z różnych programów nie uległy przemieszaniu.

Aby to zapewnić należy:

- zadania zapisu kierować do blokującego się FIFO
- bloki muszą mieć rozmiar mniejszy lub równy PIPE_BUF
system sam zadba o to, żeby dane się nie pomieszały

Kiedy proces Unixa jest zablokowany, **nie zużywa zasobów procesora**,
- **metoda synchronizacji procesów za pomocą blokujących się FIFO jest bardzo wydajna.**

zapis danych do łącza - write

int write(int fd, char* buf, int count)

- numer deskryptora (funkcja open)
- bufor z danymi do przesłania
- rozmiar przesyłanych danych

if (count < rozmiaru_łącza) - wpisanie danych wykonane przez niepodzielna operacje (zapisane w całości)

else dane dzielone na mniejsze porcje, które są umieszczane w łączy niezależnie od siebie. Może to spowodować przemieszanie danych wpisywanych przez kilka procesów

pobieranie danych z łącza - read

int read(int fd, char* buf, int count)

zamykanie deskryptora łącza – close

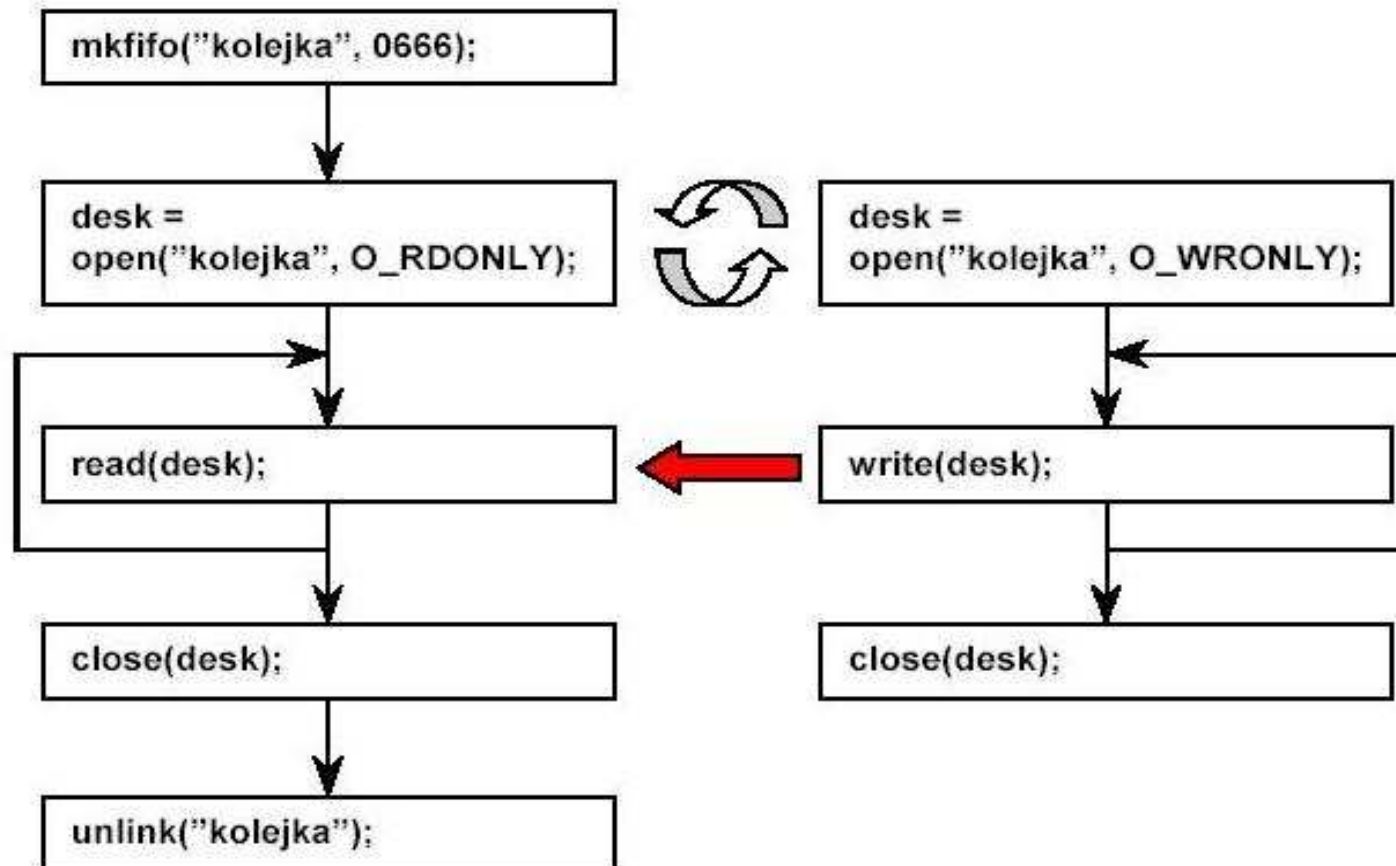
usunięcie łącza nazwanego - unlink()

int unlink(char* nazwa_lacza)

Jeżeli z łącza korzystają procesy zostaje usunięta nazwa łącza z dysku (nowe procesy nie mogą z niego korzystać). Łącze zostanie usunięte, gdy wszystkie procesy korzystające z niego zamkną deskryptory z nim związane.

Odczyt i zapis danych za pomocą funkcji: READ, WRITE, jak dla plików

Schemat komunikacji przez kolejkę FIFO:



FIFO – program klient - serwer

- serwer tworzy swoje FIFO w trybie tylko do odczytu i blokuje się
- pozostaje w tym stanie do momentu aż połączy się z nim klient, otwierając to samo FIFO do zapisu
- serwer odblokuje się i wykona sleep
- po otwarciu FIFO serwera każdy klient tworzy własne FIFO o unikatowej nazwie, przeznaczone do odczytywania danych zwracanych przez serwer
- przesyła dane do serwera (blokując się jeżeli potok jest pełny albo serwer nadal uśpiony)
- blokuje się na odczycie własnego potoku, oczekując na odpowiedź serwera
- po otrzymaniu danych od klienta serwer przetwarza je, otwiera FIFO klienta do zapisu (odblokowując w ten sposób klienta)
- zapisuje przetworzone dane
- po odblokowaniu klient może odczytać ze swojego potoku dane zapisane przez serwer

- cały proces powtarza się dopóki ostatni klient nie zamknie potoku serwera, wówczas funkcja read w serwerze zwróci 0

FIFO – kliserw.h

```
//plik naglowkowy kliserw.h
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define NAZWA_FIFO_SERWERA "/tmp/serw_fifo"
#define NAZWA_FIFO_KLIENTA "/tmp/kli_%d_fifo"
#define ROZMIAR_BUFORA 20
struct dane_do_przekazania {
    pid_t pid_klienta;
    char jakies_dane[ROZMIAR_BUFORA - 1];
};
```

FIFO – serwer.c

```
#include "kliserw.h"
#include <ctype.h>
int main()
{
    int fifo_serwera_fd, fifo_klienta_fd;
    struct dane_do_przekazania moje_dane;
    int odczyt_res;
    char fifo_klienta[256];
    char *tymczasowy_wskaznik;
    mkfifo ( NAZWA_FIFO_SERWERA, 0777 );
    fifo_serwera_fd = open ( NAZWA_FIFO_SERWERA,
    O_RDONLY );
    if ( fifo_serwera_fd == -1 ) {
        fprintf ( stderr, "Blad przy otwieraniu fifo serwera\n" );
        exit ( EXIT_FAILURE );
    }
    sleep ( 10 );
```

FIFO – serwer.c (c.d.)

```
do {
    odczyt_res = read (fifo_serwera_fd, &moje_dane, sizeof (moje_dane) );
    if ( odczyt_res > 0 ) {
        tymczasowy_wskaznik = moje_dane. jakies_dane;
        while (*tymczasowy_wskaznik ) {
            *tymczasowy_wskaznik = toupper (*tymczasowy_wskaznik);
            tymczasowy_wskaznik++;
        }
        sprintf ( fifo_klienta, NAZWA_FIFO_KLIENTA,
moje_dane.pid_klienta);
        fifo_klienta_fd=open(fifo_klienta, O_WRONLY);
        if (fifo_klienta_fd != -1)
            {write(fifo_klienta_fd, &moje_dane, sizeof(moje_dane) );
             close ( fifo_klienta_fd );}
    }
}
while ( odczyt_res > 0);
```

FIFO – serwer.c (c.d.)

```
close ( fifo_serwera_fd );  
unlink (NAZWA_FIFO_SERWERA);  
exit (EXIT_SUCCESS );  
}
```

FIFO – klient.c

```
//program klienta klient.c
#include "kliserw.h"
#include <ctype.h>
int main() {
int fifo_serwera_dp, fifo_klienta_dp;
struct dane_do_przekazania moje_dane;
int liczba_trans;
char fifo_klienta[256];
fifo_serwera_dp = open ( NAZWA_FIFO_SERWERA, O_WRONLY );
if ( fifo_serwera_dp == -1 ) {
    fprintf ( stderr, "Nie znaleziono serwera\n" );
    exit ( EXIT_FAILURE );
}
moje_dane.pid_klienta = getpid();
sprintf ( fifo_klienta, NAZWA_FIFO_KLIENTA, moje_dane.pid_klienta);
if ( mkfifo ( fifo_klienta, 0777) == -1 ) {
    fprintf ( stderr, "Nie mozna bylo utworzyc %s\n", fifo_klienta );
    exit ( EXIT_FAILURE ); }
```

FIFO – klient.c (c.d.)

```
for ( liczba_trans = 0; liczba_trans < 5; liczba_trans++ ) {
    sprintf (moje_dane. jakies_dane, "Halo z procesu %d", moje_dane.pid_klienta);
    printf ( "%d wyslal %s, ",moje_dane.pid_klienta, moje_dane. jakies_dane );
    write (fifo_serwera_dp, &moje_dane, sizeof(moje_dane) );
    fifo_klienta_dp = open ( fifo_klienta, O_RDONLY );
    if ( fifo_klienta_dp != -1 ) {
        if ( read (fifo_klienta_dp, &moje_dane, sizeof(moje_dane) ) > 0 ) {
            printf ( "Otrzymal: %s\n", moje_dane. jakies_dane ); }
        close ( fifo_klienta_dp );
    }
}
close ( fifo_serwera_dp );
unlink (fifo_klienta );
exit (EXIT_SUCCESS );
}
```

1586 wyslal Halo z procesu 1586, Otrzymal: HALO Z PROCESU 1586
1586 wyslal Halo z procesu 1586, Otrzymal: HALO Z PROCESU 1586
1586 wyslal Halo z procesu 1586, Otrzymal: HALO Z PROCESU 1586
1588 wyslal Halo z procesu 1588, Otrzymal: HALO Z PROCESU 1588
1588 wyslal Halo z procesu 1588, Otrzymal: HALO Z PROCESU 1588
1588 wyslal Halo z procesu 1588, Otrzymal: HALO Z PROCESU 1588
1588 wyslal Halo z procesu 1588, Otrzymal: HALO Z PROCESU 1588
1588 wyslal Halo z procesu 1588, Otrzymal: HALO Z PROCESU 1588
1589 wyslal Halo z procesu 1589, Otrzymal: HALO Z PROCESU 1589
1589 wyslal Halo z procesu 1589, Otrzymal: HALO Z PROCESU 1589
1589 wyslal Halo z procesu 1589, Otrzymal: HALO Z PROCESU 1589
1589 wyslal Halo z procesu 1589, Otrzymal: HALO Z PROCESU 1589
1589 wyslal Halo z procesu 1589, Otrzymal: HALO Z PROCESU 1589
1587 wyslal Halo z procesu 1587, Otrzymal: HALO Z PROCESU 1587
1587 wyslal Halo z procesu 1587, Otrzymal: HALO Z PROCESU 1587
1587 wyslal Halo z procesu 1587, Otrzymal: HALO Z PROCESU 1587
1587 wyslal Halo z procesu 1587, Otrzymal: HALO Z PROCESU 1587
1587 wyslal Halo z procesu 1587, Otrzymal: HALO Z PROCESU 1587
1590 wyslal Halo z procesu 1590, Otrzymal: HALO Z PROCESU 1590
1590 wyslal Halo z procesu 1590, Otrzymal: HALO Z PROCESU 1590
1590 wyslal Halo z procesu 1590, Otrzymal: HALO Z PROCESU 1590
1590 wyslal Halo z procesu 1590, Otrzymal: HALO Z PROCESU 1590
1590 wyslal Halo z procesu 1590, Otrzymal: HALO Z PROCESU 1590