Łącza POSIX (i nie tylko)

Ostatnia modyfikacja: 18.02.2019

POSIX

- **FIFO special file (or FIFO)**

„A type of file with the property that data written to such a file is read on a first-in-first-out” basis.

- **Pipe**

„An object accessed by one of the pair of file descriptors created by the ***pipe( )*** function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.”

Symbole związane z FIFO/pipe

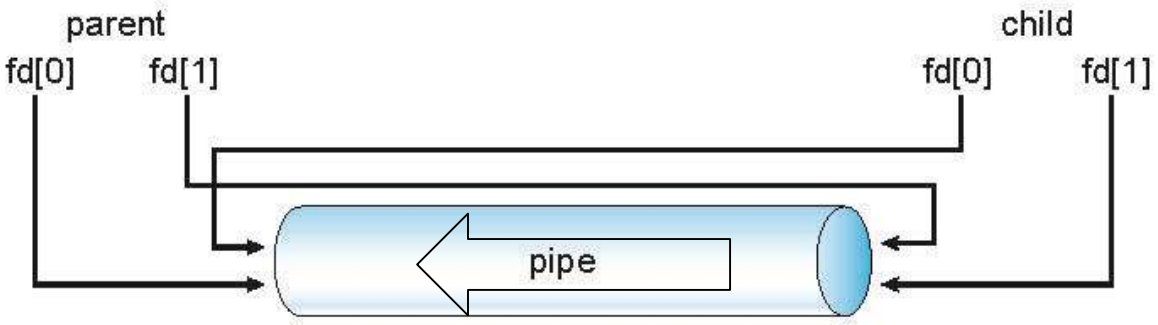
Header file	Symbols
<errno.h>	EPIPE, ESPIPE
<limits.h>, <unistd.h>	PIPE_BUF
<signal.h>	SIGPIPE
<sys/stat.h>	S_ISFIFO(*m*)

Łącza

- Łącze tworzy kanał komunikacji pomiędzy dwoma procesami
- Kwestie podstawowe:
  - Połączenie jedno- czy dwukierunkowe?
  - Half-duplex (przemienność kierunków komunikacji), czy full-duplex?
  - Czy wymagana jest jakaś specjalna relacja (n.p. rodzic-potomek) pomiędzy procesami?
  - Czy łącze może być użyte do komunikacji zdalnej (przez sieć)?
- **Łącza zwykłe (anonimowe)** – po utworzeniu przez proces nie są widoczne przez inne procesy. Dostęp może być jednak przekazywany (np. procesom potomnym).
- **Łącza nazwane** – mogą być udostępniane każdemu procesowi dzięki nazwie.

Komunikacja producent-konsument

- **Zwykłe łącza** umożliwiają realizację relacji producent-konsument.
- Producent zapisuje dane z jednego końca łącza (*the **write-end** of the pipe*)
- Konsument odczytuje dane z drugiego końca łącza (*the **read-end** of the pipe*)



- Zwykłe łącza są zazwyczaj jednokierunkowe (POSIX: tak, Windows: tak, UNIX z podsystemem STREAMS: niekoniecznie)
- Wymagane jest przekazanie deskryptorów związanych z końcami łącza:
  - Proces potomka może odziedziczyć deskryptory procesu rodzica
  - W systemach UNIX możliwe jest przekazania deskryptorów za pomocą gniazd lokalnych (*sockets*).

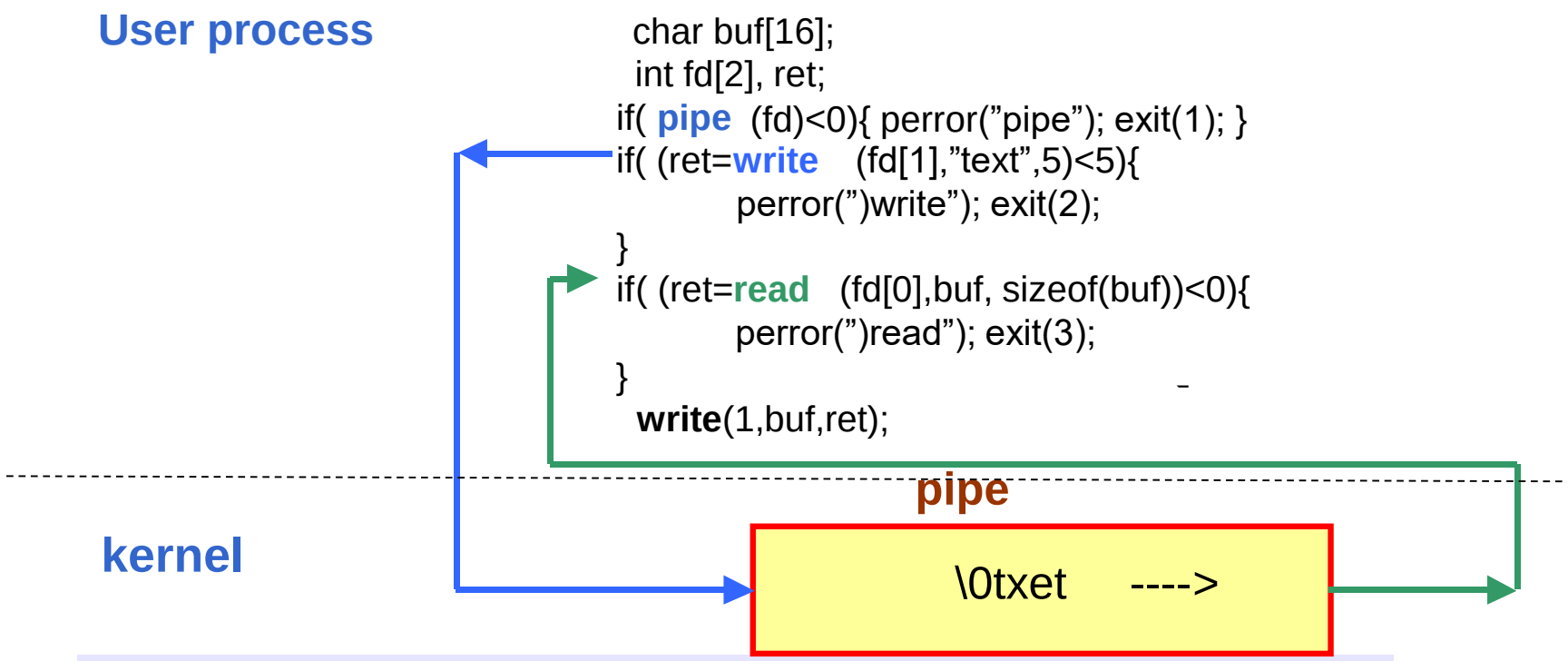
UNIX/Posix - tworzenie łącza anonimowego

```
#include <unistd.h>
int ret=pipe (int filedes[2])
```

tworzy jednokierunkowe łącze (pipe), zwracając deskryptora otwartego do odczytu do **filedes**[0] a deskryptora otwartego do zapisu do **filedes**[1]

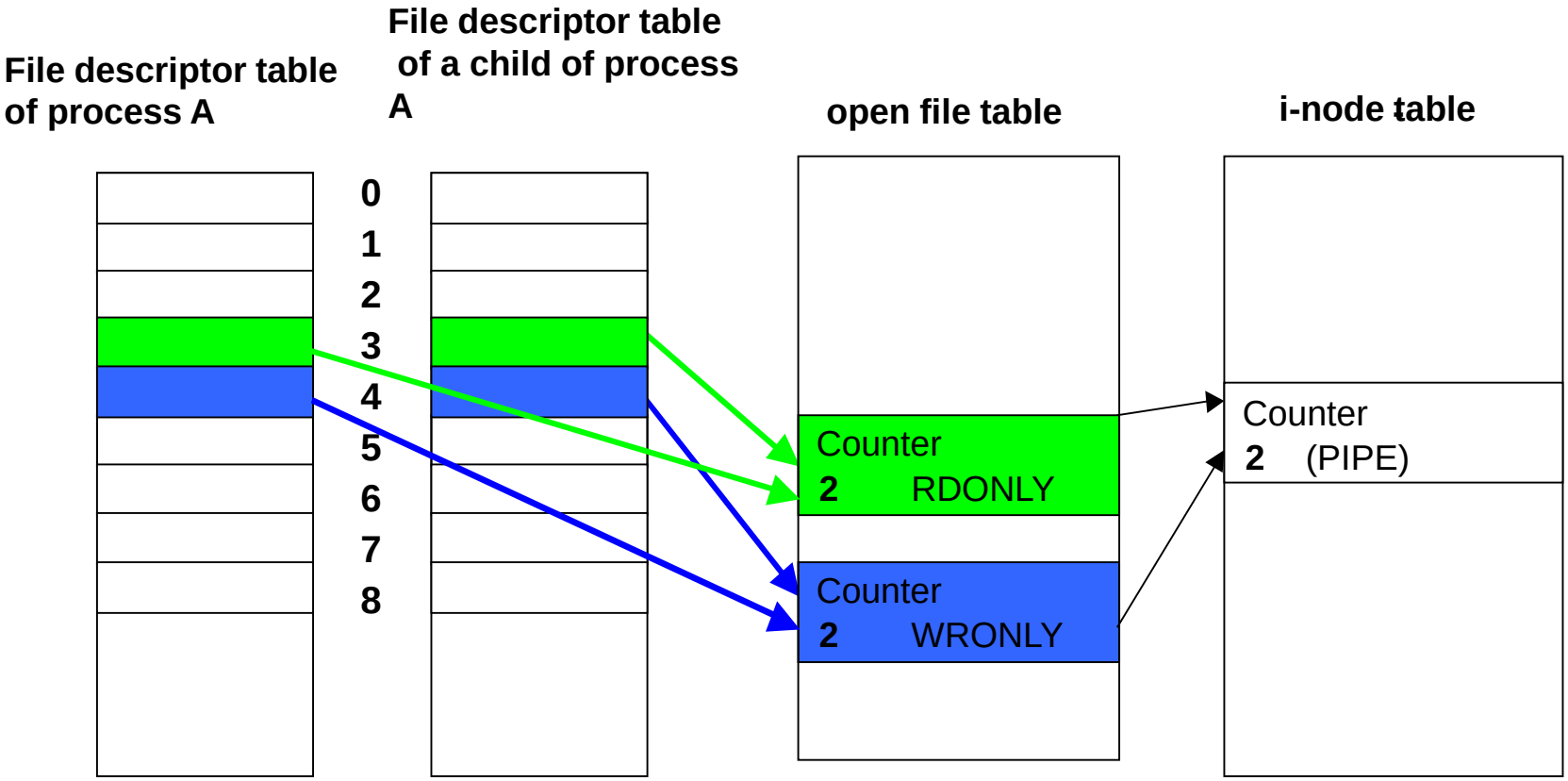
W przypadku powodzenia **ret==0**, inaczej **ret==-1** (kod błędu w **errno**).

Przykład tworzenia i trywialnego wykorzystania łącza



UNIX/POSIX: łącze i fork()

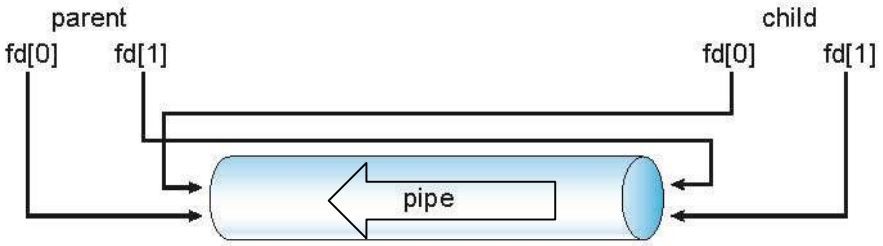
fork() udostępnia deskryptory łącza procesowi potomnemu, umożliwiając komunikację rodzic-potomek

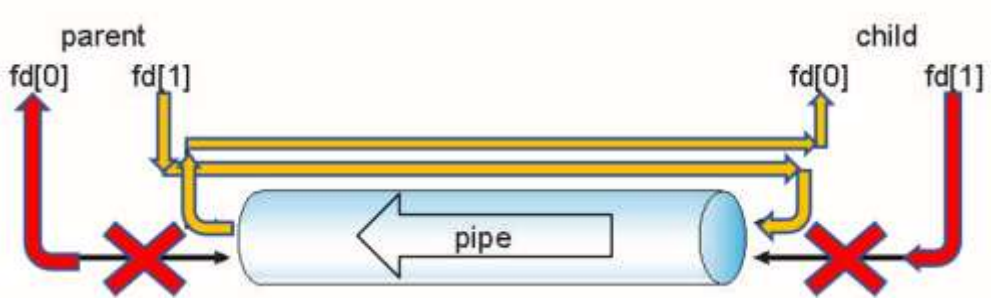


IPC za pomocą łącza anonimowego - 1

Przykład. Proces rodzica wysyła dane do procesu potomnego (**write**() / **read**()).

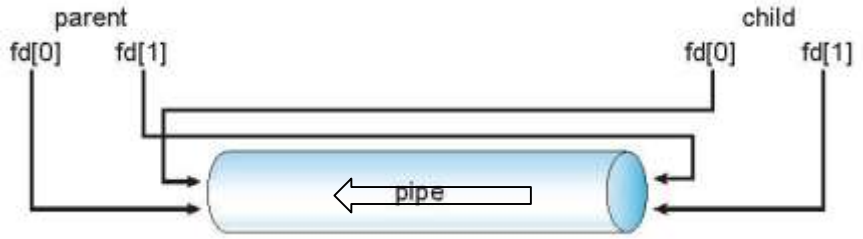
```
. . .
int main(void){
char buf[16];
int fd[2], pid, ret;
    if( pipe(fd)<0){/* error ha
    if( (pid=fork()) == -1 ){ /* error handling */ }
    else if(pid>0){ /* rodzic */
       close(fd[0]); /* zamykanie nieużywanego końca łącza */
       if( (ret=write(fd[1],"Text",5)<5){/* error */}
        /* waiting for child process */
       if( wait(NULL)<0 ){ /* error handling */ }
    } else { /* potomek */
       close(fd[1]); /* zamykanie nieużywanego końca łącza */
       if( (ret=read(fd[0],buf, sizeof(buf))<0){{/* error */}
       write(1,buf,ret);
    }
    return EXIT_SUCCESS;
}
```



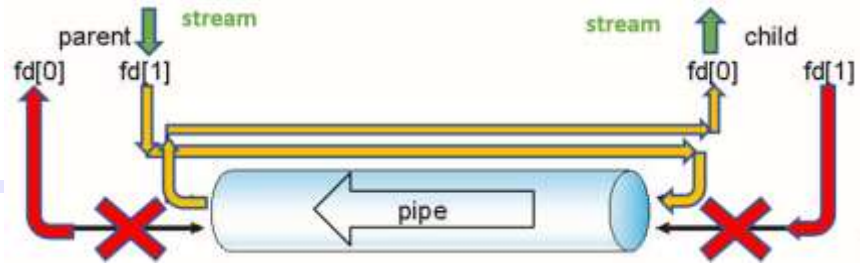


IPC za pomocą łącza anonimowego - 2

Przykład. Proces rodzica wysyła dane do procesu potomnego (przez strumień).




```
...
int main(void){
int fd[2], ret;
pid_t pid;
FILE *stream;
     if( pipe(fd)<0){/* error handling */ }
     if( (pid=fork()) == -1 ){ /* error handling */ }
     else if(pid>0){ /* rodzic */
        close(fd[0]); /* zamykanie nieużywanego końca łącza */
        stream=fdopen(fd[1],"w");
        fprintf(stream,"Message …");
        fclose(stream);
        if( wait(NULL)<0 ){ /* error handling */ }
     } else {/* potomek */
        close(fd[1]); /* zamykanie nieużywanego końca łącza */
        stream =fdopen(fd[0],"r");
        while( (ret=fgetc(stream)) != EOF )
                putchar(ret);
        fclose(stream);
     }
     return EXIT_SUCCESS;
}
```



Łącza – c.d.

Własności łącza anonimowego:

- Dostęp do łącza – tylko poprzez deskryptory (dziedziczone od procesu-twórcy łącza).
- Łącze nie wspiera pozycjonowania. Próba odczytu/ustawienia pozycji kończy się niepowodzeniem z **errno=ESPIPE** (invalid seek)
- Próba zapisu do zamkniętego łącza ustawia kod błędu **errno=EPIPE** (broken pipe); wysyłany jest też sygnał **SIGPIPE** do procesu, który podjął próbę.
- Łącze ma ograniczoną pojemność (**PIPE_BUF** >=512B)
- Łącze przechowuje sekwencję bajtów. Nie ma znaczników końca rekordów logicznych.
- Odczyt z/zapis do łącza jest nierozdzielny (*atomic*) jeśli rozmiar danych jest nie większy niż **PIPE_BUF**. Jeśli blok danych o rozmiarze <= **PIPE_BUF** przepełnia łącze – proces zapisu jest wstrzymywany – aż do uzyskania odpowiednio dużego wolnego miejsca w łączu. Bloki danych dłuższe niż **PIPE_BUF** są przesyłane we fragmentach.
- Kiedy wszystkie deskryptory związane z łączem zostaną zamknięte dane pozostałe w łączu giną (łącze też).

Łącza POSIX/UNIX– cd.

#include <stdio.h>

FILE * fp=popen (const char *cmd, const char *mode)

Tworzy podproces (przy pomocy komendy powłoki: **cmd**) . Jeżeli **mode**, to:

- **„r”** – wówczas **fp** jest strumieniem połączonym z **stdout** podprocesu
- **„w”** – wówczas **fp** jest strumieniem połączonym z **stdin** podprocesu

int pclose(FILE *fp)

Zamyka dostęp do strumienia utworzonego przez wywołanie funkcji **popen()**, czeka na zakończenie komendy powłoki (wykonywanej w podprocesie) i zwraca kod wyjścia podprocesu. Szczegóły: **man pclose.**

Przykład użycia popen()

```
/* cmdlog.c – Wykonuje podane polecenie (cmd), duplikując
  standardowe wyjście (plik log) */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
char buf[BUFSIZ];
FILE *fp1, *fp2;
size_t n;
  if(argc!=3){
       fprintf(stderr,"Usage: cmdlog  log  cmd"); exit(1);
   }
  if((fp1=fopen(argv[1],"w"))==NULL){ /* error */... }
  if((fp2=popen(argv[2],"r"))==NULL){ /* error */...}

  while((n=fread(buf,sizeof(char),sizeof(buf),fp2))>0){
       if(fwrite(buf,sizeof(char),n,fp1)!=n){/* error */...}
       if(fwrite(buf,sizeof(char),n,stdout)!=n){/* error */..}
  }
  (void) pclose(fp2); (void) fclose(fp1);
  return 0;
}
```

FIFO (łącze nazwane)

FIFO – typ pliku specjalnego, charakteryzującego się tym, że sekwencja zapisanych bajtów jest odczytana w kolejności zapisu („in the first-in-first-out order").

```
#include <sys/types.h>
#include <sys.stat.h>
int mkfifo(const char *path, mode_t mode);
```

Tworzy wpis w systemie pliku typu FIFO. Lokalizację FIFO określa ścieżka (względna lub bezwzględna) **path**, **mode** określa prawa dostępu (jak przy wywołaniu **open()**).

FIFO może być wykorzystane do komunikacji jednokierunkowej tak samo jak łącze anonimowe, jeśli tylko zostanie otwarte **zarówno** do odczytu jak i do zapisu.

Jedno łącze może być wykorzystane przez więcej niż dwa procesy (np. dwóch nadawców i jeden odbiorca komunikatów). Do komunikacji potrzebny jest przynajmniej jeden nadawca komunikatów (deskryptor z prawami zapisu) i jeden odbiorca (deskryptor z prawami odczytu).

Uwaga: Typowo jeden proces otwiera łącze do zapisu, a inny do odczytu. Domyślnie operacje otwarcia dostępu są blokujące; funkcje **open** blokują wołające wątki aż łącze nie zostanie otwarte do odczytu **oraz** do zapisu.

Przykład użycia FIFO

```
/* fifosv.c : MYFIFO -> stdout*/
…
#define FIFO_FILE "MYFIFO"
int main(void){
  char buf[80];
  int fd, m, n;
  unlink(FIFO_FILE);
  umask(0);
  if(mkfifo(FIFO_FILE, 0666)){. .}
  if((fd=open(FIFO_FILE,O_RDONLY)) <0){
    . . .
  }
  while((n=read(fd,buf,80)) >0 ){
     if((m=write(1,buf,n))!=n){ . . .
     } else {
``````fprintf(stderr,"%d B read\n”,n);
     }
   } /* while() */
   if(n==0){
````  fputs("EOD\n",stderr); return 0;
   }
   if(errno) perror(„fifosv”);
   return 0;
}
```

```
/* fifocl.c : stdin -> MYFIFO */
…
#define FIFO_FILE "MYFIFO"
void handler(int sig){
  fputs("SIGPIPE\n",stderr); return;
}
int main(int argc, char*argv[]){
   char buf[80];// what if 8000?
   int fd, m, n;
   signal(SIGPIPE,handler); /* ☺ */
   if((fd=open(FIFO_FILE,O_WRONLY))<0){.}
   while((n=read(0,buf,80))>0 ){
      if( (m=write(fd,buf,n))!=n ){…
      } else {
         fprintf(stderr,
              "%d B to FIFO\n",m);
      }
    }/* while() */
    if(n==0)fputs("EOD\n",stderr);
    else {
      if(errno==EINTR)
         fputs("EINTR\n",stderr);
      else perror("fifocl");
    }
    return 0;
}
```

Niebezpieczeństwo blokady przy otwieraniu FIFO

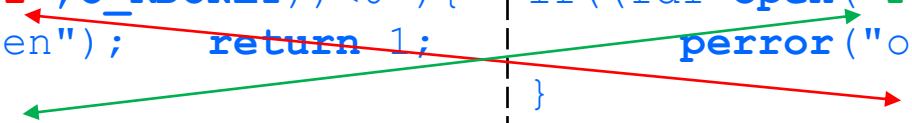
Nieprzemyślane rozpoczynanie dwukierunkowej komunikacji za pomocą dwóch FIFO może skutkować blokadą procesów. Przykład:

```
/*  prog1.c */
...

int main(void){
int fd1, fd2;
...
if((fd1=open("F21",O_RDONLY))<0 ){
      perror("open");    return 1;
}
if((fd2=open("F12",O_WRONLY))<0 ){
      perror("open");    return 1;
}
/* code which is to read from fd1
 * and to write to fd2 */
...
 return 0;
}
```

```
/*  prog2.c */
...

int main(void){
int fd1, fd2;
...
if((fd1=open("F12",O_RDONLY))<0 ){
      perror("open");    return 1;
}
if((fd2=open("F21",O_WRONLY))<0 ){
      perror("open");    return 1;
}
/* code which is to read from fd1
 * and to write to fd2  */
...
 return 0;
}
```



Nieblokujące użycie FIFO

POSIX: wywołanie funkcji ***open***

- Jeśli ustawiono sygnalizatory **O_NONBLOCK** i **O_RDONLY**, to funkcja ***open*** natychmiast powraca, zwracając -1 przy niepowodzeniu (inaczej numer deskryptora >=0).
- Jeśli ustawiono sygnalizator **O_NONBLOCK** i **O_WRONLY** funkcja ***open*** sygnalizuje błąd jeśli łącze nie jest otwierane do odczytu przez (inny) wątek/proces.
- Domyślnie sygnalizator **O_NONBLOCK** nie jest ustawiony; wywołanie funkcji ***open***
  - dla trybu tylko do odczytu (**O_RDONLY**) blokuje wołający wątek aż łącze zostanie otwarte do zapisu przez inny wątek.
  - dla trybu tylko do zapisu (**O_WRONLY**) blokuje wołający wątek aż łącze zostanie otwarte do odczytu przez inny wątek/proces."

Przykład: użycia O_NONBLOCK dla uniknięcia blokady przy otwieraniu FIFO.

```
for(i=0; i<20; i++){/* try 20 times (polling)*/
   fd = open(pname,O_RDONLY|O_NONBLOCK);/* returns -1 if failed */
   if(fd!=-1) break;
   if(errno!=ENXIO){
        perror("opening FIFO"); exit(1);
   } else {
        printf("waiting for a client"); sleep(2);
   }
}
```

Nieblokujące użycie FIFO - cd.

POSIX: Próba zapisu do łącza za pomocą funkcji ***write*** przy ustawionym sygnalizatorze **O_NONBLOCK**

- Wywołanie ***write*** nie blokuje
- Próba zapisu co najwyżej {**PIPE_BUF**} bajtów danych ma następujące skutki:
  - Jeśli jest wystarczająco dużo wolnego miejsca w łączu – funkcja ***write*** zapisze wszystkie dane do (bufora) łącza i zwróci liczbę zapisanych bajtów.
  - W przeciwnym przypadku funkcja ***write*** nie zapisuje danych w łączu, zwraca **−1** ustawiając zmienną ***errno*** na [**EAGAIN**].
- Próba zapisu więcej niż {**PIPE_BUF**} bajtów danych ma następujące skutki:
  - Jeśli przynajmniej jeden bajt może być zapisany do łącza – zapis jest zrealizowany, a funkcja ***write*** zwraca liczbę zapisanych danych.
  - W przeciwnym przypadku funkcja ***write*** nie zapisuje danych w łączu, zwraca **−1** ustawiając zmienną ***errno*** na [**EAGAIN**].

Zapis do łącza - podsumowanie

Write to a Pipe or FIFO with O_NONBLOCK *clear*			
Immediately Writable:	**None**	**Some**	***nbyte***
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

Write to a Pipe or FIFO with O_NONBLOCK *set*			
Immediately Writable:	**None**	**Some**	***nbyte***
nbyte≤{PIPE_BUF}	−1, [EAGAIN]	−1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	−1, [EAGAIN]	<*nbyte* or −1, [EAGAIN]	≤*nbyte* or −1, [EAGAIN]

Nieblokujące użycie FIFO - cd.

POSIX: Wywołanie funkcji ***read*** dla pustego łącza (nazwanego lub anonimowego):

- Jeśli żaden proces nie ma łącza otwartego do zapisu funkcja ***read*** zwraca wartość **0**, co oznacza koniec danych (end-of-file).
- Jeśli jakiś proces ma łącze otwarte do zapisu:
  - Jeśli ustawiono sygnalizator **O_NONBLOCK**, to ***read*** zwraca **−1** ustawiając zmienną ***errno*** na [**EAGAIN**].
  - Jeśli sygnalizator **O_NONBLOCK** jest wyzerowany (stan domyślny), to ***read*** blokuje wołający wątek aż w łączu pojawią się dane, albo wszystkie dostępy do łącza w trybie zapisu zostaną zamknięte.

Uwagi:

Funkcja ***read*** może zwrócić **0**

- W trybie blokującym - jeśli napotkano koniec danych
- W trybie nieblokującym (ustawione **O_NONBLOCK**) gdy łącze jest (tymczasowo) puste lub żaden proces nie jest połączony z tym łączem do zapisu. Koniec danych musi być rozpoznany przez zawartość strumienia danych.

Włączanie/wyłączanie blokującego trybu obsługi wejścia/wyjścia

```
#include <fcntl.h>
int set_nonblock_flag (int desc, int value){
/*      Set the O_NONBLOCK flag of desc file descriptor,
        if value is nonzero,
        or clear the flag if value is 0.
        Return 0 on success, or -1 on error with errno set.
        Source: glibc documentation.
*/
    int oldflags = fcntl(desc, F_GETFL, 0);
  /* If reading the flags failed, return error indication */
  if (oldflags == -1)
        return -1;
  /* Set just the flag we want to set. */
  if (value != 0)
        oldflags |= O_NONBLOCK;
  else
        oldflags &= ~O_NONBLOCK;
  /* Store modified flag word in the descriptor. */
  return fcntl(desc, F_SETFL, oldflags);
}
```

Łącza w MS Win

- **Łącza anonimowe** mogą służyć jedynie do komunikacji lokalnej.
- W MS Windows **łącza nazwane** mogą służyć do komunikacji jednostronnej bądź dwukierunkowej (*duplex*) pomiędzy procesem serwera łącza (tworzącym łącze) oraz procesami klientów łącza. Komunikacja może być lokalna, bądź pomiędzy procesami na różnych komputerach. Łącza są nazywane przez ścieżki UNC (Universal Naming Convention), np.

\\myhost\pipe\mypipe wskazuje łącze **mypipe** na komputerze **myhost**

- Instancje łącza nazwanego współdzielą jedynie nazwę; dzięki temu wielu klientów łącza może niezależnie komunikować się z serwerem łącza.
- Łącza mogą przenosić strumienie bajtów bądź komunikatów.

Uwaga: w systemie Linux (>=3.4) istnieje tryb pakietowy dla łącz anonimowych (**man pipe**), który można użyć do przesyłania komunikatów o zmiennej długości. Jeśli wywołanie funkcji **write**() zapisze *n<=PIPE_BUF* bajtów do łącza – tworzy pakiet. Wywołanie funkcje **read**() może odczytać cały ten pakiet (i tylko ten pakiet), używając odpowiednio dużego bufora (o długości >=*n*).