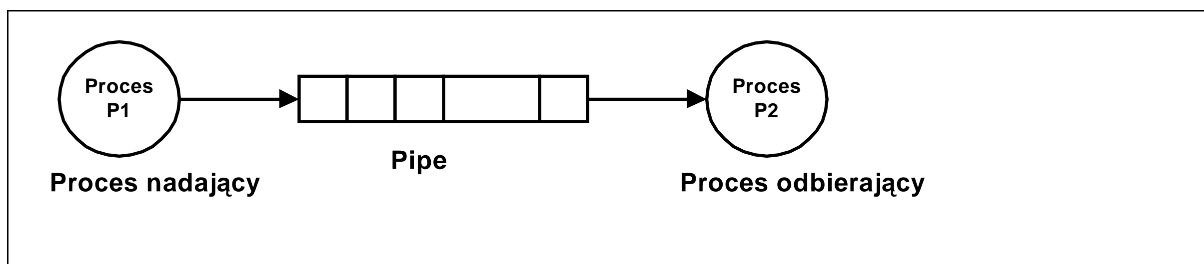
4. Komunikacja pomiędzy procesami przez łącza nienazwane i nazwane

Łącza nienazwane (*ang. Unnamed Pipes*) i nazwane (*ang. Unnamed Pipes*) - jedna z historycznie pierwszych metod komunikacji międzyprocesowej. Wywodzą się z systemu UNIX.

4.1 Łącza nienazwane

Łącze nienazwane *(ang. Pipe*) można wyobrazić sobie jako rodzaj „rury bitowej" łączącej dwa procesy.
Łącze nienazwane implementowane jest jako bufor cykliczny.



Rys. 4-1 Procesy P1 i P2 komunikują się poprzez łącze nienazwane

Łącze tworzy się poprzez wykonanie funkcji **`pipe`**:

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

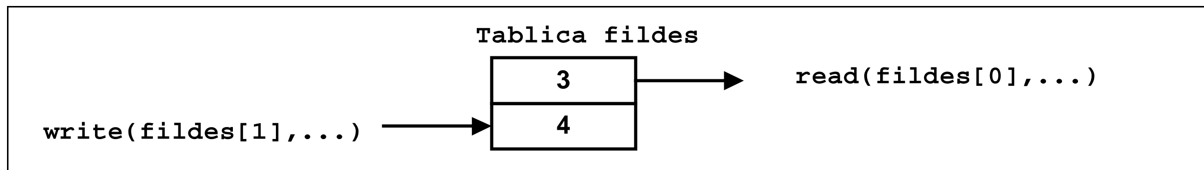
`fildes` Tablica dwuelementowa na uchwyty plików do odczytu I zapisu

Funkcja tworzy łącze nienazwane i umieszcza w tablicy fildes uchwyty plików:
`fildes[0]` – uchwyt pliku do odczytu.
`fildes[1]` – uchwyt pliku do zapisu.

Funkcja zwraca: 0 – sukces, -1 – błąd.

Do pisania i czytania z łącza używa się mechanizmu plików i standardowych funkcji **`read i write.`**
Plików nie otwiera się za pomocą funkcji **`open`**.
Stosownych uchwytów dostarcza funkcja pipe w tablicy fildes.

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 4-2 Użycie tablicy z uchwytami plików

Własności łącz nienazwanych:

1. Kanał jest jednokierunkowy dla danego procesu i nieużywany w tym procesie plik powinien być zamknięty.
2. Metoda komunikacji może być użyta tylko dla procesów związanych – będących w relacji macierzysty / potomny.
3. Jako że łącze jest buforem typu FIFO utrzymywanym w pamięci operacyjnej ma ono ograniczoną pojemność.
4. Operacje zapisu odczytu do łącza są operacjami atomowymi.

```
main() {
  int fd[2],rd,wr,i;
  char c;
  // Utworzenie łącza ---------------
  pipe(fd);
  // Utworzenie procesu potomnego ---
  if (fork() > 0) { // Proces macierzysty - czyta z łącza
     close(fd[1]);
     do {
       rd = read(fd[0], &c, 1);
       printf("Odczyt-> %c \n",c);
     } while(rd > 0);
     close(fd[0]);
  } else {  // Proces potomny - pisze do łącza -----------
       close(fd[0]);
       for(i=0;i<10;i++) {
         c= '0' + i;
         printf("Zapis-> %c \n",c);
         write(fd[1], &c,1);
         sleep(1);
       }
       close(fd[1]);
  }
}
```

Program 4-1 Przykład komunikacji dwóch procesów poprzez łącze nienazwane

PDF created with pdfFactory trial version www.pdffactory.com

Blokowanie odczytu i zapisu przy operowaniu na łączach nienazwanych

Przy posługiwaniu się mechanizmem łącz pojawiają się wątpliwości.

1. Jak zachowa się funkcja **`read`** gdy odczytywane łącze jest puste ?.
2. Jak zachowa się funkcja **`write`** gdy zapisywane łącze jest pełne ?.

O zachowaniu się procesów flaga O_NONBLOCK związana z plikami specjalnymi tworzonymi przez funkcję pipe.

Pliki te domyślnie mają wyzerowaną flagę O_NONBLOCK.
Flagę tę można kontrolować przy pomocy funkcji **`fcntl`**.
Wielkość bufora łącza można testować wykonując funkcję:

```
fpathconf(fd,_PC_PIPE_BUF)
```

	Flaga O_NONBLOCK wyzerowana	Flaga O_NONBLOCK ustawiona
Odczyt	Funkcja read blokuje proces bieżący gdy łącze puste	Funkcja read zwraca –1 gdy łącze puste
Zapis	Funkcja write blokuje proces bieżący gdy łącze pełne	Funkcja write zwraca –1 gdy łącze pełne

Wpływ flagi O_NONBLOCK na blokowanie się procesów

Flagę O_NONBLOCK testuje się przy pomocy funkcji:

```
fcntl(fd, F_GETFL, O_NONBLOCK)
```

Ustawić ją można:

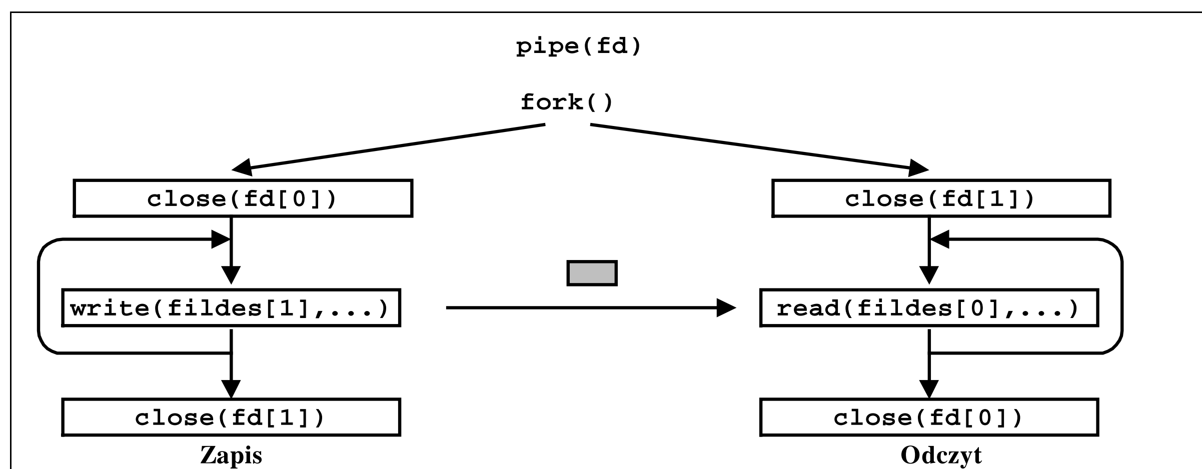
```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, O_NONBLOCK)|O_NONBLOCK)
```

Zamykanie łącz
Co stanie się gdy deskryptor reprezentujący łącze zostanie zamknięty?

1. Zamknięcie deskryptora pliku do zapisu. Gdy istnieją inne procesy które mają otwarte te łącze dla zapisu nie dzieje się nic. Gdy nie gdy nie w łączu danych procesy zablokowane na odczycie zwracają zero.
2. Zamknięcie deskryptora pliku do odczytu. Gdy istnieją inne procesy które mają otwarte te łącze dla odczytu nie dzieje się nic. Gdy nie do wszystkich procesów zablokowanych na zapisie wysłany zostanie sygnał SIGPIPE.

Rozdział 4

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 4-3 Wzorzec wykorzystania łącz nienazwanych do komunikacji pomiędzy procesami

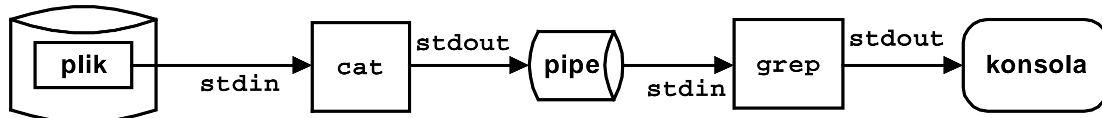
PDF created with pdfFactory trial version www.pdffactory.com

4.2 Implementacja potoków

Łącza nienazwane wykorzystywane są do implementacji potoków gdzie standard output jednego procesu kierowany jest na standard input drugiego.

Przykład

```
$ cat plik| grep tekst
```



Rys. 4-4 Wykorzystanie potoków

W mechanizmie tworzenia potoków wykorzystywane są mechanizmy:

1. Proces tworzony przez fukcje **`fork / exec`** dziedziczy stdin, stdout i strerr.
2. Jako bufor pośredni wykorzystuje się łącze pipe
3. Przekierowanie wejścia **`stdinp`** na część do odczytu łącza pipe wykonuje się funkcją **`dup2(pipe[0], 0)`**
4. Przekierowanie wyjścia **`stdout`** na część do zapisu łącza pipe wykonuje się funkcją **`dup2(pipe[1], 1)`**

Do utworzenia nowego procesu wykorzystuje się funkcję:

```
int execve(const char *filename, char *const argv[],
                   char *const envp[]);
```

`filename` Nazwa pliku wykonywalnego
`* argv[]` Lista argumentów – tablica wskaźników do łańcuchów
`* envp[]` Zmienne otoczenia - tablica wskaźników do łańcuchów

PDF created with pdfFactory trial version www.pdffactory.com

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

int main(int argc, char *arg[]) {
  int pipefd[2];
  int pid;
  char *cat_args[] = {"cat", "pipe3.c", NULL};
  char *grep_args[] = {"grep", "int", NULL};

  // Utworzenie lacza: pipefd[0]-odczyt pipefd[1]-zapis
  pipe(pipefd);
  pid = fork();
  if (pid == 0)  {
      // Zastepujemy standard input końcem do odczytu potoku
      dup2(pipefd[0], 0);
      // zamkniecie nieuzywanej czesci lacza
      close(pipefd[1]);
      // wykonanie grep
      execvp("grep", grep_args);
   } else  {
      // Zastepujemy standard output końcem do zapisu potoku
      dup2(pipefd[1], 1);
      // zamkniecie nieuzywanej czesci lacza
      close(pipefd[0]);
      // wykonanie cat
      execvp("cat", cat_args);
    }
}
```

Program 4-2 Program **pipe3** wykonuje polecenie: **cat pipe3.c | grep int** czyli w pliku **pipe3.c** znajduje linie zawierające **int** czyli program implementuje potok

PDF created with pdfFactory trial version www.pdffactory.com

```
$./pipe3
// Program wykonuje polecenie cat pipe3.c | grep int
// Czyli w pliku pipe3.c znajduje linie zawierajace int
int main(int argc, char *arg[]) {
  int pipefd[2];
  int pid;
  char *grep_args[] = {"grep", "int", NULL};
```

Przykład 4-1 Wykonanie programu pipe3

PDF created with pdfFactory trial version www.pdffactory.com

4.3 Wykorzystanie funkcji popen do tworzenia potoków

Uprzednie mechanizmy tworzenia potoków działają na poziomie mechanizmu we / wy niskiego poziomu (uchwyty). Podobny mechanizm istnieje na poziomie strumieni. Wykorzystuje funkcję popen.

```
FILE *popen(const char *command, const char *type);
```

Gdzie:

command Polecenie które ma być wykonane prze shell
type Łańcuch określający typ łącza ”r” do odczytu ”w” do zapisu

Funkcja tworzy nowy proces, nowe łącze i w kontekście nowego procesu wywołuje shell przekazując mu do wykonania polecenie ***command***. Sposób połączenia łącza określa parametr ***type***. Funkcja zwraca wskaźnik na deskryptor strumienia FILE który może być użyty w procesie macierzystym. Z/do deskryptora można czytać/pisać dane uruchomionego programu.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 80

int main() {
    int i=0;
    FILE * fp;
    char * res;
    char linia[SIZE];
    printf("Start ls\n");
    fp = popen("ls","r");
    if(fp == NULL) {
        perror("popen");
        return 0;
    }
    while(fgets(linia,SIZE,fp) != NULL) {
        printf("Plik %d %s",i,linia);
        i++;
    };
    pclose(fp);
    return 0;
}
```

Program 4-3 Proces popen_ex1 wykonuje polecenie ls przechwytując jego wyniki poprzez łącze, wykorzystana funkcja popen

PDF created with pdfFactory trial version www.pdffactory.com

```
$./popen_ex1
Start ls
Plik 0 pipe1.c
Plik 1 popen-ex1
Plik 2 pop-wyn.txt
Plik 3 wyn_pipe.txt
```

Przykład 4-2 Wynika działania programu popen_ex1

int pclose(FILE *_stream_);

Gdzie:
stream Strumień utworzony funkcją popen

Funkcja zamyka strumień utworzony funkcją popen.

PDF created with pdfFactory trial version www.pdffactory.com

4.4 Łącza nazwane – pliki specjalne typu FIFO

Pliki fifo

- tworzone są w pamięci operacyjnej,
- widziane są jednak w przestrzeni nazw plików
- posiadają zwykłe atrybuty pliku w tym prawa dostępu.

Plik FIFO tworzy się przy pomocy funkcji:

```
int mkfifo(char * path, mode_t mode)
```

path Nazwa pliku FIFO (ze ścieżką)
mode Prawa dostępu do pliku .

Funkcja zwraca: 0 – sukces, -1 – błąd.

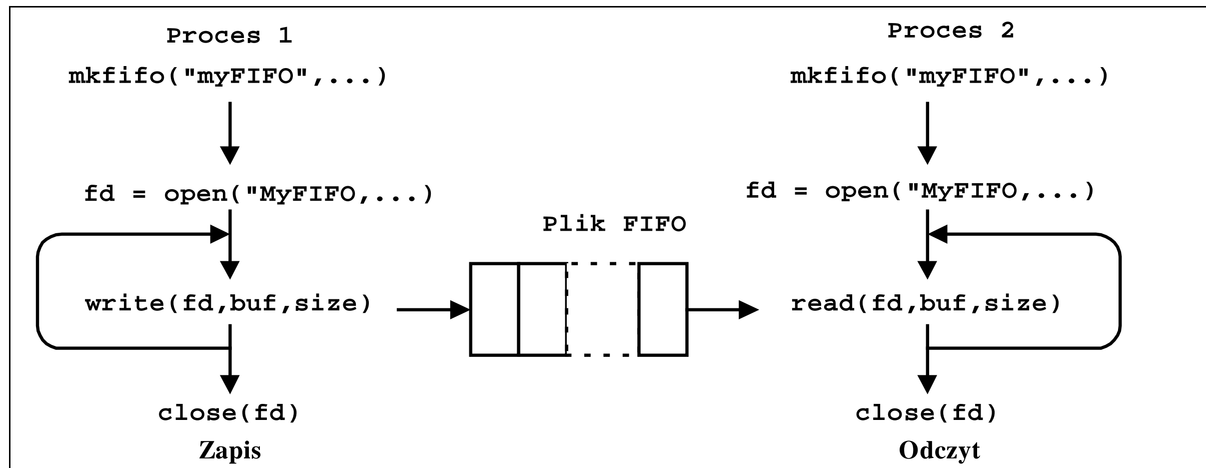
Aby proces mógł użyć pliku FIFO należy:

1. Utworzyć plik FIFO za pomocą funkcji **mkfifo** o ile wcześniej nie został utworzony.
2. Otworzyć plik FIFO za pomocą funkcji **open**.
3. Pisać lub czytać do / z pliku używając funkcji **read** lub **write**.
4. Zamknąć plik przy pomocy funkcji **close**.

Własności plików FIFO.

1. Pliki FIFO są plikami specjalnymi tworzonymi w pamięci operacyjnej ale widzianymi w systemie plików komputera. Stąd procesy mające dostęp do tego samego systemu plików mogą się komunikować przez pliki FIFO.
2. Operacje zapisu odczytu do / z pliku FIFO są operacjami atomowymi.
3. Bajty odczytane z pliku FIFO są stamtąd usuwane.
4. Zachowanie się procesu przy próbie odczytu z pustego pliku FIFO lub zapisu do pełnego zależą od flagi O_NONBLOCK.
5. Informacje w pliku FIFO są pozbawione struktury.
6. Plik FIFO i jego zawartość ginie przy wyłączeniu komputera.

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 4-5 Wzorzec wykorzystania plików FIFO

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
main() {
 int fdes,res;
 static char c;
   /* Tworzenie Lacza nazwanego */
    if(mkfifo("MyPip",S_IRUSR | S_IWUSR) < 0) {
        perror("Pipe"); return 0;
    }
    printf("Serwer task \n");
    fdes = open("MyPip",O_RDONLY);
    if(fdes < 0) { perror("Open"); return 1; }
    printf("fdes = %d \n",fdes);
    do {
      sleep(1);
      res = read(fdes, &c, 1);
      if(res < 0)  perror("reading message");
      printf("R -->%c \n", c);
    } while(res > 0);
    close(fdes);
}
```

Przykład 4-3 Przykład procesu odczytującego znaki z pliku FIFO.

PDF created with pdfFactory trial version www.pdffactory.com

```
#include <stdio.h>
...
main() {
 int fdes,res;
 static char c;
 /* Otwarcie łacza nazwanego */
 printf("Client task \n");
 fdes = open("MyPip",O_WRONLY);
 if(fdes < 0) {
     perror("Pipe"); return 0;
 }
 c = '0';
 do {
      sleep(1);
      res = write(fdes, &c, 1);
      if(res < 0)  perror("writing message");
      printf("W -->%c \n", c);
      c++;
  } while((res > 0) && (c < '9'));
  close(fdes);
}
```

Przykład 4-4 Przykład procesu zapisującego znaki do pliku FIFO.

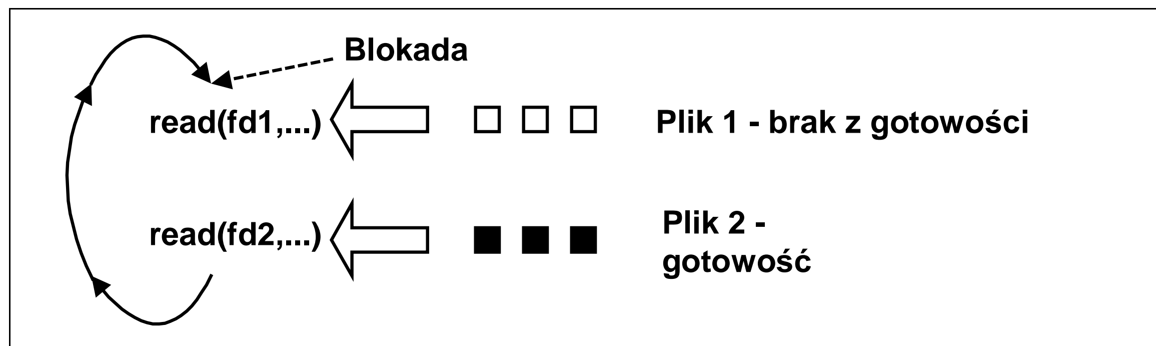
PDF created with pdfFactory trial version www.pdffactory.com

4.5 Sprawdzanie gotowości deskryptorów – funkcja select

Problem występuje gdy:

- Proces otrzymuje dane z wielu źródeł
- Odczyt blokujący

Rezultat:
Gdy oczekujemy na wejście z jednego procesu (wywołanie blokujące) nie odbieramy tego co jest na innych wejściach.



Zastosowanie – źródła dające się opisać jako deskryptory plików

1. Łącza nienazwane (pipes)
2. Łącza nazwane (pliki FIFO)
3. Gniazdka (sockets)
4. Znakowe rządzenia we/wy - klawiatura, złącza transmisji szeregowej
5. Blokowe urządzenia we/wy – pliki

PDF created with pdfFactory trial version www.pdffactory.com

Funkcja select

Funkcja select powoduje zablokowanie procesu bieżącego do czasu wystąpienia gotowości lub błędu na którymś z deskryptorów. Zwraca wtedy numer tego deskryptora. Odblokowuje się również wtedy gdy upłynie zadany okres oczekiwania (timeout).

```
int select(int nfds, fd_set *readfds,
    fd_set *writefds,fd_set *errorfds,
    struct timeval * timeout)
```

Gdzie:

nfds liczba deskryptorów plików (maksymalne FD_SETSIZE)
readfds maska deskryptorów plików do odczytu (gotowość odczytu)
writefds maska deskryptorów plików do zapisu (gotowość zapisu)
errorfds maska deskryptorów plików dotyczących błędów
timeout maksymalny okres zablokowania

Po wykonaniu funkcja ponownie ustawia maski bitowe **readfds, writefds, errorfds** zgodnie z wynikiem operacji czyli jeden bit ustawia inne zeruje.

Funkcja zwraca:

> 0 – numer deskryptora na którym wystąpiła gotowość
0 – gdy zakończenie na przeterminowaniu
-1 – gdy błąd

Funkcje operujące na maskach bitowych

fd_set - typ zdefiniowany w <sys/time.h>. Bit i ustawiony na 1 o obecności deskryptora i w zbiorze.

Przykład

Deskryptory o numerach 1,2,3 są w zbiorze

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0

- Zerowanie zbioru fdset

```
void  FD_ZERO( fd_set  *fdset)
```

- Włączenie deskryptora fd do zbioru fdset

```
void  FD_SET( int fd, fd_set  *fdset)
```

- Testowanie czy fd włączony do zbioru fdset

```
int  FD_ISSET( int fd, fd_set  *fdset)
```

Rozdział 4

PDF created with pdfFactory trial version www.pdffactory.com

• Wyłączenie fd ze zbioru fdset

```
void  FD_CLR( int fd, fd_set  *fdset)
```

Przykład

```
int fd, fd1, fd2
fd_set  we, we1;
fd1 = open("/dev/ttyS1",O_RDWR);
fd2 = open("/dev/ttyS2",O_RDWR);
FD_ZERO(&we);
FD_SET(fd1,&we);
FD_SET(fd2,&we);

do {
     we1 = we
     fd = select(5,&we1,NULL,NULL,NULL);
     if(FD_ISSET(fd1,&we1) {   read(fd1,...);    }
     if(FD_ISSET(fd2,&we1) {   read(fd2,...);    }
} while (1);
```

Przykład 4-5 Wykorzystanie funkcji select do odczytu z 2 portów szeregowych

Specyfikacja czasu

Do określenia czasu oczekiwania stosuje się zmienną typu timeval (struktura) zdefiniowaną w pliku `<sys/time.h>`.

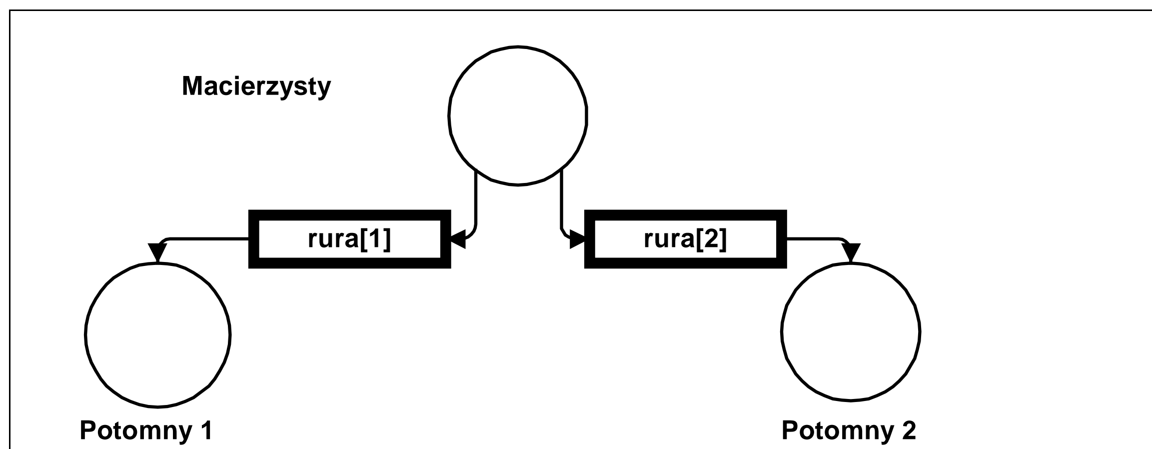
Przykład

```
#include <sys/time.h>
struct timeval {
     long tv_sec;   // sekundy
     long tv_usec;  // mikrosekundy
}

struct timeval tim;
tim.tv_sec = 10;
tim.tv_usec = 0;
```


PDF created with pdfFactory trial version www.pdffactory.com

Przykład
Serwer odczytuje zlecenia od 2 procesów klientów przez łącza nienazwane. Nie wiadomo na którym z łącz pojawi się zlecenie.



PDF created with pdfFactory trial version www.pdffactory.com

```
#include <sys/select.h>
#define SIZE 9
char msg[2][SIZE] = {"Proces 1","Proces 2"};

void main(void) {
 int rura[2][2];
 int i,pid,numer,bajtow, j = 0;;
 fd_set set;
 char buf[SIZE];

 FD_ZERO(&set);
 printf("set: %x\n",set);
 for(i=0;i<2;i++) {
    pipe(rura[i]);
    FD_SET(rura[i][0],&set);
 }

 for(i=0;i<2;i++) { // Uruchamianie procesów potomnych
   if((pid = fork(0)) == 0) {
    potom(i,rura[i]);
    exit(0);
   }
 }

 // Macierzysty -------------
 do {
  numer = select(FD_SETSIZE,&set,NULL,NULL,NULL);
  for(i=0;i<2;i++) {
       if(FD_ISSET(rura[i][0],&set)) {
            bajtow = read(rura[i][0],buf,SIZE);
       printf("Z: %d otrzymano: %s\n",i,buf);
       }
  }
  j++;
 } while(j<5);
}

int potom(int nr,int rura[2]) {
  int i;
  for(i=0;i<5;i++) {
     printf("Potomny: %d pisze: %s do: %d\n",nr, msg[nr],
     rura[1]);
     write(rura[1],msg[nr],SIZE);
     sleep(1);
  }
  return(1);
}
```


PDF created with pdfFactory trial version www.pdffactory.com