Akademia Techniczno-Humanistyczna w Bielsku-Białej

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 3

Łącza nazwane

GRUPA: 2B / SEMESTR: 5 / ROK: 3

Igor Gawłowicz / 59096

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 ...

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

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 -m a=rw /tmp/km-fifo
```

Jeżeli wykonamy teraz

```
$  ls -l /tmp/km-*
prw-rw-rw- 1 zciwolvo zciwolvo 0 Oct 26 15:41 /tmp/km-fifo
```

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   fifo
Device: 51h/81d Inode: 2594713     Links: 1
Access: (0666/prw-rw-rw-)  Uid: ( 1000/zciwolvo)   Gid: ( 1000/zciwolvo)
Access: 2023-10-26 15:41:51.685573925 +0000
```

```
Modify: 2023-10-26 15:41:51.685573925 +0000
Change: 2023-10-26 15:41:51.685573925 +0000
Birth: 2023-10-26 15:41:51.685573925 +0000
```

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, jask 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 kmfifo. 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.

input (odróżniamy od outputu za pomocą znaku >)

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

output (odróżniamy od outputu za pomocą znaku <)

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

Możemy zauważyć że po zakończeniu sesji plik pozostaje w katalogu ale jego rozmiar jest zerowy. Ponieważ łącze usuwamy jako plik.

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

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

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.

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

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)`

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

```
#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( (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));
    }
    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;
}
```

Następnie skompilujemy przykład

```
$ gcc -Wall fifo.c -o fifo
```

I uruchomimy go

```
$ ./fifo
errno=0 ...Success... mkfifo()
POZDROWIENIE (OD SAMEGO SIEBIE)
```

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 takiej sytuacji może być poniższy kod

```
#include <stdio.h>
#include <unistd.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;
}
```

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.

```
```

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

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.

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.

Server

```c
#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; }
    // 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;
}
```

Client

```
#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; }
    // 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;
}
```

Jednak aby tego użyć musimy najpierw utworzyć nasze łączę poprzez

```
$ mkfifo pipe
```

Po czym możemy zweryfikować to czy nasze łącze istnieje w poniższy sposób:

```
$ ls -l
total 180
-rwxr-xr-x 1 zciwolvo zciwolvo 17104 Oct 23 19:44 bidir
-rw-r--r-- 1 zciwolvo zciwolvo  2332 Oct 23 19:44 bidir.c
-rw-r--r-- 1 zciwolvo zciwolvo  1032 Nov  8 21:45 client.c
-rwxr-xr-x 1 zciwolvo zciwolvo 17264 Oct 23 19:37 duplicate
-rw-r--r-- 1 zciwolvo zciwolvo  2163 Oct 23 19:37 duplicate.c
-rwxr-xr-x 1 zciwolvo zciwolvo 17072 Nov  8 21:29 fifo
-rw-r--r-- 1 zciwolvo zciwolvo  1224 Nov  8 21:24 fifo.c
-rwxr-xr-x 1 zciwolvo zciwolvo 16752 Oct 23 10:07 fork
-rw-r--r-- 1 zciwolvo zciwolvo   524 Oct 23 10:06 fork.c
-rwxr-xr-x 1 zciwolvo zciwolvo 16712 Oct 23 09:54 pid
-rw-r--r-- 1 zciwolvo zciwolvo   169 Oct 23 19:01 pid.c
prw-r--r-- 1 zciwolvo zciwolvo     0 Nov  8 21:44 pipe
-rwxr-xr-x 1 zciwolvo zciwolvo 17208 Oct 23 19:30 quad
-rw-r--r-- 1 zciwolvo zciwolvo  1387 Oct 23 19:30 quad.c
-rw-r--r-- 1 zciwolvo zciwolvo   913 Nov  8 21:29 README-cloudshell.txt
-rw-r--r-- 1 zciwolvo zciwolvo  1017 Nov  8 21:45 server.c
-rwxr-xr-x 1 zciwolvo zciwolvo 16800 Nov  8 21:40 timer
-rw-r--r-- 1 zciwolvo zciwolvo   352 Nov  8 21:40 timer.c
```

Teraz możemy uruchomić nasz server oraz program kliencki i zobaczyć czy wszystko działa poprawnie

client

```
$ ./client

[7011]*K*L*I*E*N*T*[7011]

        ?...    a
        ?...    test servera
        ?...    jak widac server dziala!
        ?...
```

server

```
$ ./server

[7742]*S*E*R*W*E*R*[7742]

[7733]  |A|-> Wed Nov  8 21:51:06 2023
[7733]  |T|-> Wed Nov  8 21:51:08 2023
[7733]  |J|-> Wed Nov  8 21:51:13 2023
```

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

server

```
#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;
}
```

client

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

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

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..!.!" ); }
    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;
}
```

Teraz na potrzeby zadania stworzymy łączę channel

`$ mkfifo channel`

Następnie uruchomimy nasz server z parametrem `&`, który oznacza uaktywnienie procesu przez system.

```
$ ./server &
[1] 8796
```

Teraz widzimy że proces naszego servera pracuje pod identyfikatorem `8796`

```
ps
    PID TTY          TIME CMD
   7351 pts/6    00:00:00 bash
   8796 pts/6    00:00:00 server
   8925 pts/6    00:00:00 ps
```

Teraz możemy uruchomić klienta i sprawdzić jak działa.

```
$ ./client
test
test
test numer dwa
test numer dwa
jak widac server wysyla informacje zwrotna
jak widac server wysyla informacje zwrotn
```

Widzimy że po wpisaniu czegokolwiek z klienta server odsyła nam naszą wiadomość w następnej linii.

Teraz jedyne co nam zostało to zamknąć server.

```
$ kill -SIGKILL 8796
[1]+  Killed                  ./server
```

Wnioski

Łącza nazwane (named pipes) stanowią potężne narzędzie w komunikacji międzyprocesowej w systemach UNIX i podobnych. Są one używane do przesyłania danych między różnymi procesami, a nawet między procesami niespokrewnionymi.

Łącza nienazwane (unnamed pipes) są prostsze w użyciu, ale mają swoje ograniczenia. Jednym z głównych problemów z łączami nienazwanymi jest to, że są dostępne tylko dla procesów potomnych powstałych z jednego wspólnego procesu nadrzędnego. To oznacza, że nie można ich używać do komunikacji między niespokrewnionymi procesami.

W przeciwieństwie do tego, łącza nazwane mają szereg zalet:

Są identyfikowane przez nazwę i mogą być używane przez dowolny proces (nawet niespokrewniony), o ile ma odpowiednie uprawnienia. Posiadają organizację FIFO (First In First Out), co oznacza, że dane są odbierane w kolejności, w jakiej zostały wysłane. Posiadają dowiązanie w systemie plików jako plik specjalny urządzenia, aż do momentu jawnego usunięcia. Mimo że zachowują cechy pliku, są jedynie "okienkiem" do pamięci jądra systemu, co pozwala na efektywną komunikację między procesami. Łącza nazwane i nienazwane można tworzyć zarówno z poziomu interfejsu użytkownika jak i poprzez wywołania odpowiednich funkcji API systemu. W przypadku łączy nazwanych, można je tworzyć za pomocą polecenia mkfifo, natomiast w kodzie programu można użyć funkcji mkfifo().

Przy korzystaniu z łączy nazwanych warto zwrócić uwagę na operacje otwarcia i zamknięcia. Deskryptor łącza można uzyskać za pomocą funkcji open(). Domyślnie otwarcie łącza ma charakter blokujący, ale można użyć maski O_NONBLOCK, aby uniknąć blokowania w przypadku braku innych procesów gotowych do współpracy.

Łącza nazwane stanowią ważne narzędzie w programowaniu wieloprocesowym, pozwalając na skuteczną komunikację między różnymi procesami. Warto zrozumieć ich zalety i ograniczenia oraz umieć je odpowiednio wykorzystać w projektach programistycznych.