Eduards Šarņeckis
Automātika un datortehnika 2. grupa

4. atskaite

Starpprocesu komunikācija ar LPC-2478-STK izstrādes plati

Teorētiskā daļa

Starpprocesu komunikācija ir metožu kopa, kas ļauj realizēt datu apmaiņu starp vairākiem pavedieniem viena vai vairāku procesoru ietvaros. Procesi var darboties ar viena datora vai vairākiem datoriem tīklā. Starpprocesu komunikācijas metodes izšķir, balstoties uz joslas platumu, pavedienu komunikācijas latentumu un pārsūtāmo datu veida.

Starpprocesu komunikācijas veids	**Īss veida apraksts**	**Realizējams**
Fails	Datu ieraksts, kas tiek glabāts uz diska un kurām var piekļūt pēc nosaukuma ar jebkuru procesu.	Vairākās operētājsistēmās
Signāls	Sistēma ziņojums, kas tiek sūtīts no viena procesa uz citu, kas parasti netiek izmantotas, lai nodotu informāciju, bet nodod komandas.	Vairākās operētājsistēmās
Ligzda	Datu plūsma, kas parasti tiek veidota starp procesiem, kas atrodas uz tīkla datoriem vai vienā datorā.	Vairākās operētājsistēmās
Ziņojumu rinda	Atsevišķā datu plūsma, kas ir līdzīga kanālam, bet dati tiek glabāti un sūtīti pakešu veidā.	Vairākās operētājsistēmās
Kanāls	Divvirzienu datu plūsmas saskarnē ar standarta ieejam un izejam. Dati tiek nolasīti nolasa simbols pēc simbola.	Visās POSIX sistēmās, Windows
Definētais kanāls	Kanāls, kas ir realizēts izmantojot failu, nevis izmantojot ieejas un izejas.	Visās POSIX sistēmās, Windows

Semafors	Vienkārša veida struktūra, kas sinhronizē pavedienus vai procesus, kas darbojas, izmantojot vienu resursu.	Visās POSIX sistēmās, Windows
Koplietojamā atmiņu	Vairāki procesi spēj piekļūt vienai un tai pašā atmiņai. Tas nodrošina, ka visi procesi spēj mainīt datus atmiņā un uztvert datu izmaiņas, kuras ir veikuši citi procesi.	Visās POSIX sistēmās, Windows
Ziņojumu sūtīšana	Līdzīgs ziņojumu rindai	MPI, Java RMI, CORBA, DDS, MSMQ, MailSlots.
Atmiņai piekārtots fails	Atmiņa tiek saistīta ar failu, kura saturu var mainīt, piekļūstot atmiņas adresēm tiešā veidā, nevis plūsmas veidā. Metode ir līdzīga parastam failam.	Visās POSIX sistēmās, Windows

Pastāv vairāki iemesli, kas rosināja starpprocesu komunikācijas rašanos:

- Procesu un pavedienu priekšrocības noteikšana;
- Informācijas apmaiņa;
- Pielietošanas ērtība;
- Modularitāte;
- Skaitļošanas procesa paātrināšana.

Linux operētājsistēmas vidē ir iespējami veidot starpprocesu komunikāciju vairākos veidos. Šī darba ietvaros praktiskajā daļā tiks apskatīta 2 starpprocesu komunikācijas veidi, kas darbojas Linux operētājsistēmā: kanāls un koplietojamā atmiņa. Starpprocesu komunikācija izmantojot ligzdas, jau tika apskatīta iepriekšējā laboratorijas darba gaitā.

Praktiskā daļa

Kanāls (pipe)

Lai varētu uzrakstīt programmu C programmēšanas valodā, kas realizēs starpprocesu komunikāciju kanāla veidā uz viena datora, ir nepieciešams pieslēgt iesākumfailus:

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

Abi iesākumfaili definē vairākas struktūras, funkcijas un konstantes, kas ir nepieciešamas, lai nodrošinātu starpprocesu komunikāciju.

```
int     fd[2], nbytes;
pid_t   childpid;
char    string[] = "Hello, world!\n";
char    readbuffer[80];
pipe(fd);
```

Ir nepieciešams definēt mainīgos:

- fd[2] – kanāla ieejas un izejas stāvokļi;
- nbytes – glabās nolasītos datus;
- string[] – ir simbolu virkne, kas tiks pārsūtīta izmantojot kanālu;
- readbuffer[80] – ir nolasāmo datu buferis.
- pipe(fd) – izveido jaunu starpprocesu komunikācijas kanālu.

I

```
if((childpid = fork()) == -1)
{
   perror("fork");
   exit(1);
}
```

` If bloks pārbauda vai nav notikusi kļūda, konfigurējot kanālu. Ja tiek konstatēts, ka kanālu nevar izmantot komunikācijai, tad tiek izvadīts kļūdas paziņojums un programmas darbība tiek pārtraukta.

```
if(childpid == 0)
        {
            /* Child process closes up input side of pipe */
                close(fd[0]);

          /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
```

```
        else
        {
            /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes      =      read(fd[0],      readbuffer,
sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }
```

Datu sūtīšana no viena procesa uz otru notiek sekojošā veidā:

1. Ja dati nav nodoti (childpid == 0), tad pirmais process aizver kanāla ieeju, padod simbolu virkni uz kanāla izeju un pabeidz procesu;
2. Ja dati ir nodoti, tad otrais process aizver kanāla izeju un simbolu virkne tiek nolasīta no kanāla.

Koplietojamā atmiņā

Koplietojamā atmiņā ir starpprocesu komunikācijas metode, kas piedāvā izmantot kopējo atmiņas segmentu. Viena programma izveido atmiņas segmentu, kurām citi procesi varēs piekļūt.

Lai varētu uzrakstīt programmu C programmēšanas valodā, kas realizēs starpprocesu komunikāciju koplietojamās atmiņas veidā uz viena datora, ir nepieciešams pieslēgt iesākumfailus:

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

Šie iesākumfaili definē vairākas konstantes un struktūras, kas ir nepieciešamas, lai varētu izmantot starpprocesu komunikāciju (`ipc.h`) un tās konkrēto metodi – koplietojamo atmiņu (`shm.h`).

```
key = 5678;
```

Serveris definē atmiņas segmentu.

```
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

Definētais atmiņas segments tiek izdalīts. Tiek veikta pārbaude, kas segmenta izdalīšanas kļūdas gadījumā izvada kļūdas paziņojumu un pabeidz programmas darbību.

```
s = shm;
for (c = 'a'; c <= 'z'; c++)
     *s++ = c;
*s = NULL;
```

Izdalītajā atmiņā tiek ierakstīta simbolu virkne no ‘a’ līdz ‘z’, kuru pēc tam varēs nolasīt klienta programma.

Atmiņas segments tiek atrasts. Tiek veikta pārbaude, kas segmenta identificēšanas kļūdas gadījumā izvada kļūdas paziņojumu un pabeidz programmas darbību.

```
while (*shm != '*')
        sleep(1);
```

Serveris gaida, kamēr pirmais no simboliem, kas tika ierakstīti koplietojamajā atmiņā, tiks mainīts uz ‘*’, un beidz darbību.

Klienta programmas uzdevums ir nolasīt ierakstīto definētājā atmiņas segmentā informāciju.

```
key = 5678;
```

Atmiņas segmenta adrese ir tā, kas tika uzdota servera programmā.

```
if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

Tagad klienta programma meklē izdalīto atmiņas segmentu. Ja tāds segments nav definēts, tad tiek izvadīts paziņojums par kļūdu un programma beidz darbību.

```
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
```

Atmiņas segments tiek piekārtots klienta programmas datu laukam. Ja atmiņas segmentu nav iespējams piekārtot datu laukam, tad tiek izvadīts paziņojums par kļūdu un programma beidz darbību.

```
for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');
```

Klienta programma nolasa datus no koplietojamās atmiņas un izvada tos uz ekrāna.

```
*shm = '*';
```

Pirmais no simboliem, kurus satur koplietojamā atmiņa tiek izmainīts uz ‘*’.

Secinājumi

Ceturtā laboratorijas darba gaitā uz LPC-2478-STK izstrādes plates tika palaistas programmas, kas realizē starpprocesu komunikāciju.

Kanāla metode izmanto vienu programmu, kuras ietvaros tiek veidoti divi procesi, kas attiecīgi kontrolē datu padošanu un kanāla ieeju un datu nolasīšanu un kanāla izeju. Savukārt koplietojamās atmiņas ietvaros tiek izmantotas divas atsevišķas programmās: serveris, kas izdala atmiņas segmentu un ieraksta tajā informāciju, un klients, kas nolasa datus no izdalītā atmiņas segmenta.

Darba rezultātā tika iegūtas papildus zināšanas par starpprocesu komunikāciju. Izmantojot C programmēšanas valodu, Linux vidē tika praktiski realizētas kanāla metode un koplietojamās atmiņas metode un palaistas uz LPC-2478-STK izstrādes plates.

1. pielikums

Kanāla programmas pirmkoda avots:
http://www.tldp.org/LDP/lpg/node11.html#SECTION00722000000000000000

Kanāla programmas pirmkods:

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
            /* Child process closes up input side of pipe */
                close(fd[0]);

          /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
            /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

2. Pielikums

Koplietojamās atmiņas programmas pirmkoda avots:
http://www.cs.cf.ac.uk/Dave/C/node27.html

shm_server programmas pirmkods:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put some things into the memory for the
     * other process to read.
     */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

shm_client programmas pirmkods:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';
    exit(0);
}
```