ДОДАТКОВА ЛАБОРАТОРНА РОБОТА №8*.

**виконується групами КІ-307, КІ-308 та КІ-309 у разі несвоєчасного виконання основних лабораторних робіт*

Укладач і автор зразку коду до завдання: Козак Н.Б., ст. викладач.

Автор відомостей про сокети Берклі:** Бочкарьов О.Ю., доцент, к.т.н..

*** посібник «Паралельне програмування в ОС Linux» / Бочкарьов О. Ю.: https://eom.lpnu.ua/textbooks/np-pposlinux.pdf*

ВСТУП ДО МЕРЕЖНОГО ПРОГРАМУВАННЯ: СОКЕТИ БЕРКЛІ ТА ЗМІШАНЕ ПРОГРАМУВАННЯ МОВАМИ C/C++ ТА АСЕМБЛЕРА

Мета: познайомитися з принципами мережного програмування за допомогою сокетів Берклі та застосувати їх для створення програм, частини яких написані різними мовами програмування.

ТЕОРЕТИЧНІ ВІДОМОСТІ***

**** більш детально в матеріалах наступних курсів*

HTML***

HTML (англ. HyperText Markup Language — мова розмітки гіпертексту) — стандартизована мова розмітки документів для перегляду вебсторінок у браузері. Браузери отримують HTML документ від сервера за протоколами HTTP/HTTPS або відкривають з локального диска, далі інтерпретують код в інтерфейс, який відображатиметься на екрані монітора. Елементи HTML є будівельними блоками сторінок HTML. За допомогою конструкцій HTML, зображення та інші об'єкти, такі як інтерактивні форми, можуть бути вбудовані у візуалізовану сторінку. HTML надає засоби для створення структурованих документів, позначаючи структурну семантику тексту, наприклад заголовки, абзаци, списки, посилання, цитати та інші елементи. Елементи HTML окреслені тегами, написаними з використанням кутових дужок. Теги на кшталт <img /> чи <input /> безпосередньо виводять вміст на сторінку. Інші теги, такі як <p>, оточують текст і надають інформацію про нього, а також можуть включати інші теги як піделементи. Браузери не показують теги HTML, але використовують їх для інтерпретації вмісту сторінки.

МЕРЕЖНА МОДЕЛЬ OSI ТА НАБІР ПРОТОКОЛІВ TCP/IP***

Модель OSI (EMBBC) (базова еталонна модель взаємодії відкритих систем, англ. Open Systems Interconnection Basic Reference Model, 1978 р.) — абстрактна мережева модель для комунікацій і розроблення мережевих протоколів. Представляє рівневий підхід до мережі. Кожен рівень обслуговує свою частину процесу взаємодії. Завдяки такій структурі спільна робота мережевого обладнання й програмного забезпечення стає набагато простішою, прозорішою й зрозумілішою. На сьогодні основним використовуваним стеком протоколів є TCP/IP, розроблення якого не було пов'язане з моделлю OSI і до того ж було здійснено до її прийняття. За увесь час існування моделі OSI вона не була реалізована, і, очевидно, не буде реалізована ніколи. Сьогодні використовується тільки деяка підмножина моделі OSI. Вважається, що модель занадто складна, а її реалізація займе забагато часу.

HTTP***

HTTP — протокол передачі даних, що використовується в комп'ютерних мережах. Назва скорочена від HyperText Transfer Protocol, протокол передачі гіпертекстових документів

HTTP належить до протоколів моделі OSI 7-го прикладного рівня.

Основним призначенням протоколу HTTP є передача вебсторінок (текстових файлів з розміткою HTML, зображеннь та застосунків), проте за його допомогою успішно передаються й інші файли (в цьому плані HTTP складає конкуренцію складнішому FTP).

HTTP припускає, що клієнтська програма — веббраузер — здатна відображати гіпертекстові вебсторінки та файли інших типів у зручній для користувача формі. Для правильного відображення HTTP дозволяє клієнтові дізнатися мову та кодування символів вебсторінки, запитати версію сторінки з потрібною мовою чи в потрібному кодуванні, використовуючи позначення зі стандарту MIME.

СОКЕТИ БЕРКЛІ (BERKELEY SOCKETS)

Взаємодія процесів за допомогою сокетів Берклі

Сокети Берклі (Berkeley sockets) – це механізм взаємодії процесів (IPC), що, в загальному випадку, виконуються на різних обчислювальних вузлах комп'ютерної мережі. Для програміста цей механізм представлений у вигляді відповідного програмного інтерфейсу. Метою створення сокетів Берклі було спрощення організації обміну даними між обчислювальними процесами через мережу, за рахунок абстрагування від деталей обміну мережними повідомленнями транспортного рівня OSI (протокол IP).

Механізм сокетів Берклі був розроблений в Університеті Каліфорнії у Берклі (University of California at Berkeley), і був вперше реалізований у UNIX-подібній операційній системі BSD Unix 4.2 (1983 р.). Механізм виявився настільки вдалим, що з часом став де-факто стандартом мережного програмного забезпечення. На даний час цей механізм реалізований в усіх UNIX-подібних операційних системах, зокрема в ОС Linux, а також з деякими відмінностями у ОС сімейства Windows. Механізм сокетів Берклі лежить в основі переважної більшості сучасних Інтернет-технологій та використовується майже в усіх програмах, робота яких пов'язана з Інтернетом.

Слово "socket" можна дослівно перекласти, як "гніздо роз'єму". Відповідно у сокетах Берклі використовується абстракція мережних роз'ємів, за допомогою яких один процес підключається до іншого, після чого вони обмінюються даними по встановленій лінії зв'язку. Абстракція реалізує модель клієнт-сервер. Процес, який очікує на підключення, називається сервером. Процес, який підключається до іншого процесу, називається клієнтом. Після встановлення з'єднання, забезпечується повноцінний дуплексний зв'язок між клієнтом та сервером.

В схемі взаємодії процесів за допомогою сокетів Берклі (рис.2.93) паралельно виконуються процес сервера та процес клієнта. Процес сервера створює серверний сокет (socket()), призначає йому адресу (bind()), визначає довжину черги очікування на підключення (listen()) і переходить в режим очікування підключень (accept()). Процес клієнта створює клієнтській сокет і намагається підключитись до серверного сокету за вказаною адресою (connect()). Після успішного підключення, в процесі сервера створюється комунікаційний сокет (його дескриптор повертає accept()). В такий спосіб утворюється "лінія зв'язку" між сервером та клієнтом з двома "роз'ємами": комунікаційним сокетом на стороні сервера та клієнтським сокетом на стороні клієнта. В цьому з'єднанні відбувається двонаправлений обмін даними між клієнтом та сервером (send()/recv() або write()/read()). Після завершення роботи з сокетом він знищується викликом close().

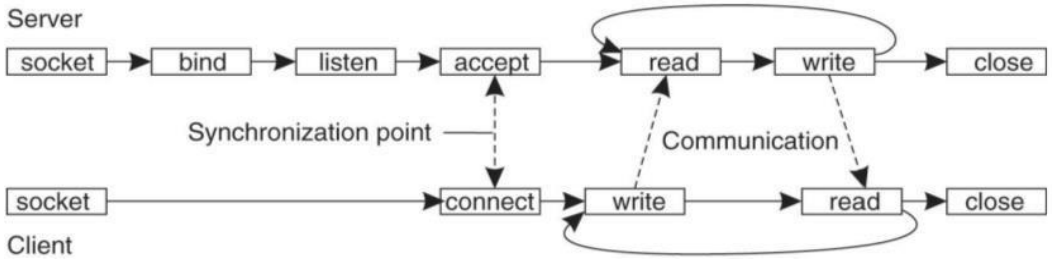

Рис.2.93. Схема взаємодії процесів за допомогою сокетів Берклі.

В механізмі сокетів Берклі підтримуються два основних режими взаємодії процесів (domains):

1) локальна взаємодія з відображенням сокетів на файлову систему (AF_UNIX або AF_LOCAL – домен системи UNIX), коли процеси клієнта та сервера виконуються локально на одному обчислювальному вузлі;

2) взаємодія процесів в мережі (AF_INET, AF_INET6 – домен Internet), коли процеси клієнта та сервера виконуються на різних обчислювальних вузлах комп'ютерної мережі (стек протоколів TCP/IP).

У випадку взаємодії процесів за допомогою сокетів в мережі використовують два основних типи сокетів:

1) віртуальний канал (stream sockets, SOCK_STREAM), коли мережна взаємодія відбувається за протоколом TCP (Transmission Control Protocol) (рис.2.94);

2) дейтаграмний зв'язок (datagram sockets, SOCK_DGRAM), коли мережна взаємодія відбувається за протоколом UDP (User Datagram Protocol) (рис.2.95).

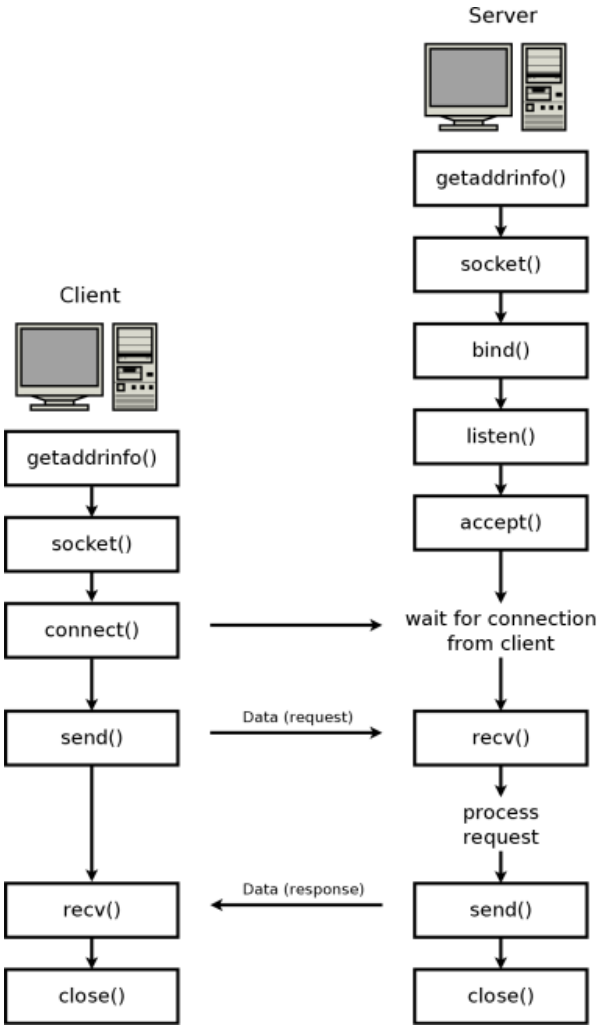

Рис.2.94. Схема взаємодії процесів за допомогою віртуального каналу (stream sockets, TCP).

Структура адреси сокету

Є три ситуації коли в програмах, що використовують сокети, відбувається робота з адресою сокета:

1) Призначення (прив'язка) адреси серверному сокету в процесі сервера за допомогою виклику bind(). В даному випадку потрібно, наприклад, дізнатись ip-адресу комп'ютера, на якому виконується програма, та зберегти цю інформацію у відповідному полі структури інтернет-адреси серверу.

2) Підключення до серверу зі сторони процесу клієнта за допомогою виклику connect(). В даному випадку, наприклад, потрібно коректно вказати адресу віддаленого сервера у структурі інтернет-адреси, що передається як параметр виклику connect().

3) В разі успішного підключення клієнта до сервера виклик accept() повертає дескриптор відповідного комунікаційного сокету, а у параметрі addr – інтернет-адресу клієнта, з яким встановлено зв'язок. В разі необхідності цю адресу можна проаналізувати і відповідним чином змінити хід роботи програми (наприклад, встановити той чи інший режим обміну даними з клієнтом або розірвати встановлене з'єднання).

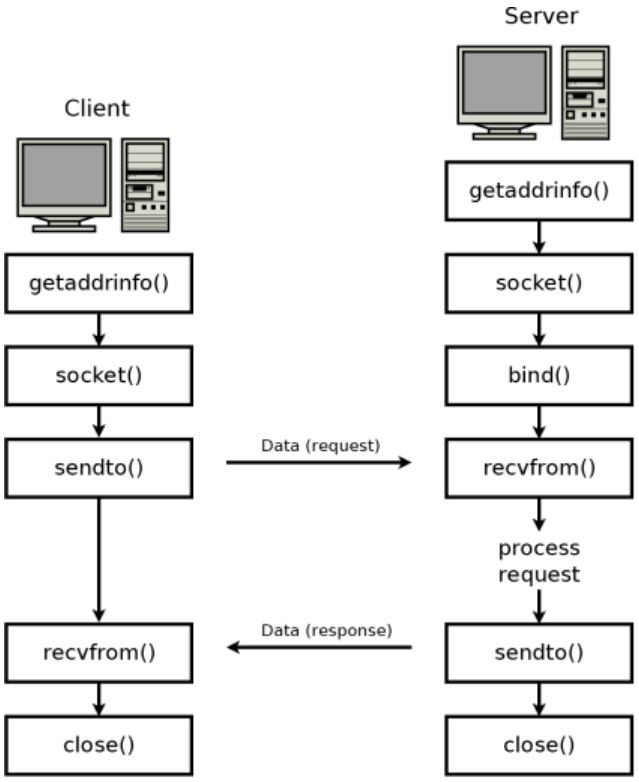

Рис.2.95. Схема взаємодії процесів за допомогою дейтаграмного зв'язку (datagram sockets, UDP).

Універсальна узагальнена структура адреси сокету sockaddr (рис.2.96) використовується для коректного визначення параметрів викликів bind(), accept(), connect(). В програмі на її місце підставляється конкретна структура адреси, яка відповідає типу сокету (локальний або мережний, IPv4 або IPv6 і т.п.). Найбільш часто це такі структури:

1) struct sockaddr_un (рис.2.97) – адреса сокету для локальної взаємодії з відображенням сокетів на файлову систему: sun_family=AF_UNIX, поле sun_PATH містить повний шлях до файлу;

2) struct sockaddr_in (рис.2.98) – адреса мережного сокету IPv4: sin_family=AF_INET, поле sin_port містить номер порту, поле sin_addr містить ip-адресу у форматі IPv4 (у вигляді структури in_addr);

3) struct sockaddr_in6 (рис.2.99) – адреса мережного сокету IPv6: sin_family=AF_INET6, поле sin6_port містить номер порту, поле sin6_addr містить ip-адресу у форматі IPv6 (у вигляді структури in6_addr).

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

Рис.2.96. Структура sockaddr.

```
struct sockaddr_un {
    short    sun family;    /* AF UNIX */
    char     sun PATH[108]; /* path name */
};
```

Рис.2.97. Структура sockaddr_un (AF_UNIX, AF_LOCAL).

```
struct in_addr {
    unsigned long s_addr;      // that's a 32-bit int (4 bytes), load with inet_pton()
};

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
```

Рис.2.98. Структура sockaddr_in (IPv4, AF_INET).

```
struct in6_addr {
    unsigned char   s6 addr[16];   // load with inet pton()
};

struct sockaddr_in6 {
    u_int16_t       sin6 family;   // address family, AF INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6 flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
```

Рис.2.99. Структура sockaddr_in6 (IPv6, AF_INET6).

Для роботи зі структурою адреси мережного сокету використовуються наступні функції:

1) inet_pton() – перетворює IP-адресу в нотації цифр і крапок у структуру in_addr або структуру in6_addr залежно від того, що вказано: AF_INET або AF_INET6 ("pton" означає "presentation to network") (рис.2.100);

2) inet_ntop() – перетворює struct in_addr або struct in6_addr в IP-адресу в нотації цифр і крапок ("ntop" означає "network to presentation") (рис.2.101);

3) getpeername() – повертає мережну адресу вузла, підключеного до сокету sockfd, у буфері, на який вказує addr (рис.2.102).

4) getaddrinfo() – визначити адресу комп'ютера для використання у функціях bind() чи connect() (рис.2.103);

5) getnameinfo() – визначити назву комп'ютера за його адресою (рис.2.104).

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

Рис.2.100. Приклад використання функції inet_pton().

```
// IPv4:
char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
```

```
printf("The IPv4 address is: %s\n", ip4);


// IPv6:
char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

Рис.2.101. Приклад використання функції inet_ntop().

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

  int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Рис.2.102. Опис функції getpeername().

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

int getaddrinfo(const char *node,     // (in) e.g. "www.example.com" or IP
                const char *service,  // (in) e.g. "http" or port number
                const struct addrinfo *hints, // (in)
                struct addrinfo **res); // (out)
```

Рис.2.103. Опис функції getaddrinfo().

```
       #include <sys/socket.h>
       #include <netdb.h>

       int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,
                       char *host, socklen_t hostlen,
                       char *serv, socklen_t servlen, int flags);
```

Рис.2.104. Опис функції getnameinfo().

Два основних способи використання функції getaddrinfo(): 1) дізнатися адресу вузла для створення серверного сокету в процесі сервера, який виконується на цьому вузлі (тобто у виклику bind()); 2) дізнатися адресу віддаленого вузла для підключення до нього з процесу клієнта (тобто у виклику connect()). В першому випадку в параметрі node вказується NULL. В другому випадку в параметрах node та service задаються інтернет-адреса вузла та інтернет-сервіс відповідно. Адреса повертається в параметрі res у вигляді структури addrinfo (рис.2.105). Оскільки адрес може бути декілька, res – це покажчик на масив структур addrinfo. В параметрі hints можна вказати попередньо заповнену структуру addrinfo, значення полів якої специфікують (обмежують) тип адреси, яка повернеться в параметрі res. На рис.2.106 наведено приклад використання функції getaddrinfo().

```
           struct addrinfo {
               int              ai_flags;
               int              ai_family;
               int              ai_socktype;
               int              ai_protocol;
               socklen t        ai_addrlen;
               struct sockaddr *ai addr;
```

```
            char            *ai_canonname;
            struct addrinfo *ai_next;
        };
```

Рис.2.105. Структура addrinfo.

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

Рис.2.106. Приклад використання функції getaddrinfo() для визначення адреси вузла, на якому виконується програма.

Створення сокету

Для створення нового програмного сокету використовується функція socket() (рис.2.107). Значення параметра domain визначає домен даного сокету (AF_UNIX – UNIX-домен або AF_INET – Internet-домен), параметр type вказує тип створюваного сокета (SOCK_STREAM – віртуальний канал (TCP) або SOCK_DGRAM – дейтаграмний зв’язок (UDP)), а значення параметра protocol визначає бажаний мережний протокол. Зауважимо, що якщо значенням параметра protocol є нуль, то система сама обирає відповідний протокол для комбінації значень параметрів domain і type (це найбільш поширений спосіб використання функції socket()). Функція socket() повертає значення дескриптора сокету, яке використовується у всіх наступних функціях. Виклик функції close(sd) призводить до закриття (знищення) зазначеного сокету.

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

int socket(int domain, int type, int protocol);
```

Рис.2.107. Опис функції socket().

Для призначення адреси раніше створеному сокету використовується функція bind() (рис.2.108). Параметр sockfd – це дескриптор раніше створеного сокета; addr – адреса структури, яка містить адресу сокета, що відповідає вимогам домену даного сокету (зокрема, для домену UNIX адреса – це ім'я об'єкту файлової системи, і при створенні сокету дійсно створюється файл); параметр addrlen містить довжину в байтах структури addr (цей параметр необхідний, оскільки довжина імені може суттєво відрізнятися для різних комбінацій "домен-

протокол"). Приклад використання функцій socket() та bind() для створення нового сокету наведено на рис.2.109.

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Рис.2.108. Опис функції bind().

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);


// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(AF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

Рис.2.109. Приклад використання функцій socket() та bind().

Функція listen() (рис.2.110) призначена для інформування системи про те, що процес серверу планує встановлення віртуальних з'єднань через вказаний сокет. Ця функція зазвичай викликається після функцій socket і bind. Вона повинна викликатися перед викликом функції accept. Параметр sockfd – це дескриптор існуючого сокету, а значенням параметра backlog є максимальна довжина черги запитів на встановлення з'єднання, які повинні буферизуватись системою, поки їх не опрацює процес-сервер.

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

int listen(int sockfd, int backlog);
```

Рис.2.110. Опис функції listen().

Встановлення з’єднання

Функція accept() (рис.2.111) використовується для отримання на опрацювання процесом сервера чергового підключення до вказаного серверного сокету. Параметр sockfd задає дескриптор серверного сокету, для якого раніше була виконана функція listen(); параметр addr вказує на структуру з адресою віддаленого клієнта, підключення якого відбулося; addrlen –

адреса, за якою знаходиться довжина структури addr. Якщо на момент виклику функції accept() черга запитів на підключення порожня, то виконання процесу сервера призупиняється (блокується) до надходження запиту. Виконання функції accept() призводить до встановлення віртуального з'єднання. Функція accept() повертає дескриптор комунікаційного сокету, який в подальшому використовується на стороні сервера для взаємодії з клієнтом для обміну даними по встановленому з'єднанню. Як правило, виклик accept() розміщують у нескінченному циклі "детектування" нових підключень, а для кожного чергового з'єднання створюють новий програмний потік, в який передають дескриптор комунікаційного сокету. Функція accept4() працює так само як accept() з тою відмінністю, що у параметрі flags можна вказати прапорці, які задають той чи інший режим роботи функції. Наприклад, за допомогою прапорця SOCK_NONBLOCK роботу accept4() можна перевести в асинхронний (неблокуючий) режим.

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

int accept4(int sockfd, struct sockaddr *addr,
            socklen_t *addrlen, int flags);
```

Рис.2.111. Опис функцій accept() та accept4().

За допомогою функції connect() (рис.2.112) процес клієнта скеровує системі запит на підключення до віддаленого вузла з вказаною адресою (тобто до серверного сокету в процесі сервера). Параметри мають те саме значення, що у функції bind(), проте в якості адреси addr вказується адреса серверного сокету (віддаленого вузла), який знаходиться на іншій стороні каналу зв'язку. Для нормального виконання функції необхідно, щоб у сокету з дескриптором sockfd і у серверного сокету з адресою addr були однакові домен і протокол. Якщо тип сокету з дескриптором sockfd є дейтаграмним (UDP), то функція connect() служить лише для інформування системи про адресу призначення пакетів, які в подальшому будуть надсилатися за допомогою функції send(); ніякі дії по встановленню з'єднання в цьому випадку не виконуються. В разі успіху, коли з'єднання встановлено, функція connect() повертає нуль. В разі помилки connect() повертає -1 (змінна errno містить код помилки). При використанні функції connect(), як правило, потрібно передбачити спроби повторного з'єднання.

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

int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Рис.2.112. Опис функції connect().

Відправлення та отримання повідомлень

Для відправлення та отримання даних через сокети зі встановленим віртуальним з'єднанням (SOCK_STREAM) використовуються функції send() і recv() (рис.2.113). В функції send() параметр sockfd задає дескриптор існуючого сокету з встановленим з'єднанням; параметр buf вказує на буфер з даними, які потрібно відправити через канал зв'язку; параметр len задає довжину цього буфера; параметр flags містить прапорці. За допомогою прапорця MSG_OOB можна відправити дані у канал в режимі "позачергового" (out-of-band) відправлення. В такому випадку дані надсилаються окремо, "обганяючи" усі раніше непрочитані з каналу дані. Потенційний отримувач даних може отримати спеціальний сигнал і в ході його обробки негайно прочитати "позачергові" дані. Функція send() повертає кількість

реально відправлених байтів даних, яка в нормальних ситуаціях збігається зі значенням параметру len.

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Рис.2.113. Опис функцій send() та recv().

В функції recv() параметр sockfd задає дескриптор сокету з встановленим з'єднанням; параметр buf вказує на буфер, в який слід помістити отримані дані; параметр len задає максимальну довжину цього буфера. Вказавши у параметрі flags прапорець MSG_PEEK, можна прочитати дані з системного буферу (socket receive bufer) в користувацький буфер buf без їх видалення з системного буферу. Функція recv() повертає кількість реально отриманих в buf байтів даних.

Зауважимо, що в разі використання сокетів з віртуальним з'єднанням замість функцій send і recv можна використовувати звичайні файлові системні виклики read() і write(). Для сокетів цього типу вони виконуються абсолютно аналогічно функціям send() і recv(). Це дозволяє створювати програми, які не залежать від того, чи працюють вони зі звичайними файлами, каналами (pipes, named pipes) або сокетами.

Для відправлення та отримання даних в дейтаграмному режимі (SOCK_DGRAM) використовуються функції sendto() та recvfrom() (рис.2.114). Параметри sockfd, buf і len аналогічні за змістом до відповідних параметрів функцій send() і recv(). Параметри dest_addr і addrlen функції sendto() задають адресу серверного сокету (віддаленого вузла), якому відправляються дані, і можуть бути опущені, якщо до цього викликалася функція connect(). Параметри src_addr і addrlen функції recvfrom() дозволяють процесу сервера отримати адресу вузла, від якого отримані дані, надіслані процесом клієнта.

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

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Рис.2.114. Опис функцій sendto() та recvfrom().

Розрив з'єднання та знищення сокету

Після завершення роботи з сокетом, потрібно закрити відповідний дескриптор за допомогою функції close(), тим самим повідомивши систему про звільнення цього системного ресурсу. Тобто викликом close(sfd) ми закриваємо і знищуємо сокет з дескриптором sfd. Відповідне мережне з'єднання розривається. У випадку віртуального каналу (stream sockets, TCP) перед знищенням сокету система спробує відправити дані, які вже поставлені в чергу, і після того як відправлення відбудеться, здійснить нормальну послідовність дій по завершенню TCP-з'єднання.

Для негайної ліквідації (розриву) встановленого з'єднання без знищення сокету використовується функція shutdown() (рис.2.115). Виклик shutdown() дозволяє негайно зупинити обмін даними в одному з трьох режимів, які задаються значенням параметра how:

1) SHUT_RD – забороняє приймання даних через сокет,

2) SHUT_WR – забороняє відправлення даних через сокет,

3) SHUT_RDWR – забороняє приймання та відправлення даних через сокет.

Дія функції shutdown() відрізняється від дії функції close() тим, що, по-перше, виконання останньої "затримується" до закінчення спроб системи завершити доставку вже відправлених даних. По-друге, функція shutdown() розриває з'єднання, але не ліквідує дескриптори раніше з'єднаних сокетів. Для їх остаточного знищення все одно потрібно викликати функцію close().

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

int shutdown(int sockfd, int how);
```

Рис.2.115. Опис функції shutdown().

КОНТРОЛЬНІ ПИТАННЯ

1. Для чого призначений механізм сокетів Берклі (Berkeley sockets)?
2. Який тип сокетів Берклі використовується для організації мережної взаємодії процесів за протоколом TCP?
3. Для чого використовується системний виклик socket() і яке значення від повертає?
4. Чим відрізняються серверний, комунікаційний та клієнтський сокети?
5. Чим відрізняється взаємодія процесів з використанням сокетів в режимі віртуального каналу (SOCK_STREAM), режимі дейтаграмного зв'язку (SOCK_DGRAM) та у режимі локальної взаємодії (AF_LOCAL)?
6. Який системний виклик ОС Linux використовується для інформування системи про те, що процес серверу планує встановлення віртуальних з'єднань через вказаний сокет із заданою максимальною довжиною черги запитів на встановлення з'єднання?
7. В якому порядку використовуються системні виклики та функції при роботі з серверним сокетом в режимі віртуального каналу (stream sockets)?
8. В якому порядку використовуються системні виклики та функції при роботі з клієнтським сокетом в режимі віртуального каналу (stream sockets)?

ЛІТЕРАТУРА

1. Джордейн Р.Справочник програмиста персональных компъютеров типа IBM PC XT и AT. - М."Финансы и статистика",1992,стор.13-31.
2. Березко Л.О., Троценко В.В. Особливості програмування в турбо-асемблері. - Київ, НМК ВО, 1992.
3. Дао Л. Программирование микропроцессора 8088.Пер. с англ. -М.: "Мир",1988.
4. Абель П.Язык ассемблера для IBM PC и программирования. Пер. з англ.- М.,"Высшая школа",1992.
5. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A & 2B): Instruction Set Reference, A-Z. – 2011. Режим доступу: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.html
6. Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear, The Art of Multiprocessor Programming, 2nd Edition, Morgan Kaufmann, 2020. – 576 p.
7. Peter Pacheco, Matthew Malensek, An Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann, 2021. – 450 p.
8. Bertil Schmidt Jorge Gonzalez-Dominguez Christian Hundt Moritz Schlarb, Parallel Programming: Concepts and Practice, Morgan Kaufmann, 2017. – 416 p.
9. Programming Models for Parallel Computing, ed. by Pavan Balaji, The MIT Press, 2015. - 488 p.
10. Michael McCool, James Reinders, Arch Robison, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012. – 432 p.
11. Thomas Rauber, Gudula Rünger, Parallel Programming For Multicore and Cluster Systems, Springer, 2010. – 455 p.
12. Fayez Gebali, Algorithms and Parallel Computing, John Wiley & Sons, 2011. – 365 p.
13. Victor Alessandrini, Shared memory application programming, Morgan Kaufmann, 2016. – 528 p.
14. W. Richard Stevens, UNIX Network Programming, Volume 2: Interprocess Communications, 2nd ed., Prentice Hall, 1998. – 558 p.
15. W. Richard Stevens, UNIX Network Programming (Volume 1): Networking APIs: Sockets

and XTI, Prentice Hall, 1998. – 1009 p.
16. Michael Kerrisk, The Linux Programming Interface: A Linux and UNIX System Programming Handbook, 1st Edition, No Starch Press, 2010. – 1553 p.
17. W. Stevens, Stephen Rago, Advanced Programming in the UNIX Environment, 3rd Edition, Addison-Wesley Professional, 2013. – 1032 p.
18. Robert Love, Linux System Programming, 2nd ed., O'Reilly Media, 2013. – 456 p.
19. Kaiwan N. Billimoria, Hands-On System Programming with Linux, Packt Publishing, 2018. - 794 p.
20. Ulrich Drepper, Ingo Molnar, The Native POSIX Thread Library for Linux, White paper, Red Hat, Inc., 2005.
21. Rohit Chandra et al. Parallel Programming in OpenMP, Academic Press, Morgan Kaufmann Publishers, 2001. – 249 p.
22. Barbara Chapman, Gabriele Jost, Ruud van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2008. – 353 p.
23. Ruud Van Der Pas, Eric Stotzer, Christian Terboven, Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD, The MIT Press, 2017. – 392 p.
24. Timothy G. Mattson, Yun (Helen) He, Alice E. Koniges, The OpenMP Common Core, The MIT Press, 2019. – 320 p.
25. OpenMP Application Programming Interface Specification Version 5.2, November 2021. – 649 p.
26. James Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core processor Parallelism, O'Reilly Media, 2007. – 336 p.
27. Michael Voss, Rafael Asenjo, James Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, Apress, 2019. – 820 p.
28. Hagit Attiya, Jennifer Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2 ed., John Wiley & Sons, 2004. – 414 p.
29. George Em Karniadakis, Robert M. Kirby II, Parallel Scientific Computing in C++ and MPI, Cambridge University Press, 2003. - 696 p.

ЗАВДАННЯ

1. Створити *.exe програму, яка реалізовує обчислення, заданого варіантом виразу. Програма повинна складатися з двох модулів:
   головний модуль – створюється мовою C(або C++) і має забезпечити html-інтерфейс(для вводу і виводу даних) за допомогою http-протоколу та виклик асемблерної процедури для обчислення виразу;
   модуль безпосередніх обчислень – здійснює всі необхідні арифметичні дії з використанням математичного співпроцесора.
2. Переконатися у правильності роботи кожного модуля зокрема та програми загалом.
3. Скласти звіт про виконану роботу з приведенням тексту програми та коментарів до неї.
4. Дати відповідь на контрольні запитання.

Таблиця 7.1.

Варіанти завдань

(відповідають варіантам додаткової лабораторної роботи №6)

№	Вираз	K
1.	$X=A_4+B_2*C_1-D_2/E_1+K$	1254021
2.	$X=A_4/B_2+C_3-D_1*E_1-K$	202
3.	$X=K-B_4+C_2/D_1-E_1*F_2$	37788663
4.	$X=A_1*(B_2+C_1)-D_1/E_2+K$	45694
5.	$X=A_2*B_2-A_2*C_1-D_1/E_2+K$	505
6.	$X=K+B_2/C_1-D_2*F_2-E_1$	6DD02316
7.	$X=A_4/B_2-C_1*(D_1+E_2-K)$	717
8.	$X=A_2-B_1+K-D_2/E_1+F_1*B_2$	88
9.	$X=A_1*B_2-A_1*C_2+D_2/(E_1+K)$	29
10.	$X=A_2-B_1/C_2+K+E_2*F_1$	2310
11.	$X=(A_2-B_1-K)*D_1+E_4/F_2$	311
12.	$X=K+B_1/C_2-D_2*F_2-E_1$	7055E0AC
13.	$X=A_2/B_1+C_1*(D_1+E_2-K)$	2513
14.	$X=A_2-B_1-K-D_2/E_1+F_1*B_1$	614
15.	$X=A_2+(B_1*C_2)-D_4/E_2+K$	4569600F
16.	$X=A_1/B_2+C_3-D_1*E_1+K$	616
17.	$X=A_1-K+C_1/D_2-E_1*F_1$	1017
18.	$X=A_1*(B_2-C_1)+D_2/E_1+K$	56987018
19.	$X=A_2*B_2+A_2*C_1-D_2/E_1+K$	4019
20.	$X=K+B_1/C_2-D_1*F_1-E_2$	18932020
21.	$X=A_1/B_2+C_1*(D_2-E_1+K)$	21
22.	$X=K-B_2-C_1-D_2/E_1+F_2*B_1$	45781022
23.	$X=A_2*B_2-C_1+D_1/E_2+K$	7AA02023
24.	$X=K-B_2/C_1+D_1+E_2*F_2$	74569024
25.	$X=(K-B_2-C_1)*D_1+E_4/F_2$	2B05025
26.	$X=A_2+K+C_2/D_1-E_1*F_1$	6C26
27.	$X=A_2*B_1+C_1/(K-E_1*F_1)$	A77627
28.	$X=K+B_1/C_2+D_2-E_2/F_1$	3FF28
29.	$X=K-B_1*C_1+D_2-F_2/E_1$	12A0C029
30.	$X=K+B_2-D_2/C_1+E_1*F_2$	25630

A, B, C, D, E, F - дійсні числа, де індекс визнає точність внутрішнього подання (1 – одинарної, 2, 3, 4 – подвійної точності). Константу К подано у 16-му форматі.

ПРИКЛАД КОДУ

Програма з взаємодією C-ASM, що виконує завдання згідно варіанту №30 розміщена за посилапнням:

https://github.com/KozakNazar/srcserver__c_asm/tree/master/srcserver__c_asm/LLNW_extended .