Лабораторна робота “Сокети”

Мета: Опанувати низкорівневий інтерфейс роботи з сокетами**.**

Задачі: Розглянути створення простого серверу та клієнта.

Завдання:

Номер індивідуального завдання обчислюється за формулою:

```
(< номер завдання >) = (< свій номер у списку > - 1) % 6 + 1
```

Свій номер у списку дивись на останній сторінці поточного документу (Додаток 1).

1. Створити однозадачний сервер (та програму клієнт до нього), який на отриманий від клієнта запит на підключення відповідає кліенту повідомленням, яке містить ip-адресу компьютера, з якого прийшов запит.

2. Створити однозадачний сервер (та програму клієнт до нього), який на отриманий від клієнта запит на підключення запитує клієнта його ім’я, після чого відповідає вітанням у форматі: “Hello, <и’мя клієнта>!”

3. Створити однозадачний сервер (та програму клієнт до нього), який після підключення до нього кліенту відправляє клієнту зміст текстового файлу, ім’я якого клієнт надсилає серверу. У випадку відсутності такого файлу, сервер інформує клієнта відповідним повідомленням.

4. Створити багатозадачний сервер (та програму клієнт до нього), який дозволяє одночасне підключення багатьох клієнтів. Кожному з підключених клієнтів сервер повідомляє його ip адресу та порядковий номер підключення.

5. Створити багатозадачний реверсний-ехо-сервер, який дозволяє одночасне підключення багатьох клієнтів. Сервер отримавши повідомлення від клієнта відправляє повідомлення у реверсному порядку символів.

6. Створити однозадачний сервер (та програму клієнт до нього), який після підключення клієнта переходить у режим діалога з клієнтом: клієнт надсилає повідомлення, сервер відправляє це повідомлення, перетворивши усі символи у верхній регістр. Діалог завершується після отримання сервером повідомлення “exit”.

Сокети

Сокети – це механізм міжпроцесної взаімодії, який дозволяє забеспечити передачу данних між процесами як одної так і різних операційних систем.

Протокол – це система правил взаімодії між двома сокетами.

При створенні сокету нам потрідно визначитись в межах якого комуникаційного домену він буде працювати. Існує декілька комунікаційних доменів. наприклад: **AF_INET**, **AF_UNIX**. Комунікаційний домен візначає можливі протоколи передачі даних між процесами. Домен **AF_UNIX** визначае протоколи взаімодії в межах однієї системи, в той час, як домен **AF_INET** забеспечую взаімодію процесів різних операційних систем.

В межах одного комунікаційного домену можно створити сокети з різними можливостями та характером взаімодії. Розглянемо створення сокету домену **AF_INET**.

В межах цього комуникаційного домену можно створити сокети різних типів, тут розглянемо тільки два з них: **SOCK_DGRAM** та **SOCK_STREAM**. Перший тип сокету забеспечує передачу даних без попереднього встановлення зв’язку від одного процесу до іншого та необхідності підтверження факту отримки цих данних, у той час як другий тип сокету вимагає таке підтверження. Тому вважається, що сокети першого типу забеспечують не надійну передачу данних (на рівні ОС), у той час, як сокету другого типу вважаються надійними.

Створення сокету

Створити сокет може тільки ядро операційної системи. Для цього існує системній виклик **socket()**:

```
#include <sys/socket.h>
int socket(domain, type, protocol);
```

Ця функція повертає файловий дескриптор сокету або -1, якщо сокет створити не вдалося. Файловий дескриптор далі буде ідентифікувати цей сокет для інших функцій.

Приклад:

```
pid_t sid; // Файловий дескриптор
if ((sid = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
```

```
        printf("Socket error\n");
        exit(1);
    }
```

Параметр **protocol** приймає значення **0**, що дає змогу системі самостійно визначитись з можливим протоколом для вказанного домену та типу сокету. У нашому випадку ядро оберає **`IPPROTO_TCP`**. (значення **6**). Інколи, у деяких версіях Linux, у якості параметру **protocol** дозволяється вказати **`IPPROTO_IP`**, що має значення **0**. Але краще вказати значення **0**, щоб не вводити в оману читача тексту програми.

Якщо помилки не сталося, то подальша робота залежить від ролі сокету: **сервер** або **кліент**. Сервер – це програма обо пристрій, який чекає запит на підключення від кліента, отримавши його обробляє цей запит і відправлє відповідь. Після чого переходить у стан очикування наступного запиту. Клієнт – це програма або пристрій, який формує запит до сервера, відправлє його і чекає від сервера відповідь. Отримавши відповідь представлє її відповідному для поточного сервісу форматі.

У цій лабораторній роботі ми будемо розглядати роботу потокового сокету **SOCK_STREAM** комунікаційного домену **AF_INET**.

Доповнення

Щоб налаштувати параметри сокету звернемося до системного виклику **setsockopt()**:

`setsockopt()` - це системний виклик, який використовується в мережевому програмуванні (зазвичай у мовах **C/C++**) для налаштування параметрів сокета. Він дозволяє змінювати різні опції сокета - як на рівні **TCP/UDP**, так і на рівні **IP** або самого сокета.

Прототип функції:

```
int setsockopt(
      int sockfd,            // дескриптор сокета
      int level,             // рівень протоколу (SOL_SOCKET, IPPROTO_TCP, тощо)
      int optname,           // ім'я опції, яку встановлюємо
      const void *optval,    // вказівник на значення опції
      socklen_t optlen       // розмір значення (в байтах)
);
```

Повертає: **`0`** при успішному виконанні; **`-1`** при помилці (тоді в **`errno`** встановлюється код помилки).

Параметри:

`sockid` Дескриптор вже створеного сокета

`Level` Рівень, до якого належить опція. Наприклад:

- **`SOL_SOCKET`** загальні опції сокета
- **`IPPROTO_TCP`** TCP-рівень
- **`IPPROTO_IP`** IP-рівень

`optname` Конкретна опція, яку хочете встановити, наприклад:

- **`SO_REUSEADDR`** Дозволяє повторно використовувати адресу і порт
- **`SO_KEEPALIVE`** Вмикає перевірку «життєздатності» з'єднання
- **`SO_REUSEPORT`** **Д**ає можливість **створювати кілька сокетів, що прив'язані до одного і того ж порту (і адреси)**, без конфліктів. У цьому випадку ядро саме визначить якому сокету направити отримані дані. Створений саме для того, щоб **рівномірно розподіляти навантаження між процесами або потоками**, які **слухають один і той самий порт**.

`optval` Вказівник на значення опції (наприклад, **`int optval = 1;`**)

`optlen` Розмір змінної, на яку вказує **`optval`**

Наприклад:

```
int opt = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
setsockopt(sockfd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));
```

Зауваження: функція **setsockopt()** дозволяє за одне звернення встановлювати значення тільки однієї опції.

Про опцію **`SO_KEEPALIVE`**:

`SO_KEEPALIVE` відповідає за перевірку "життєздатності" з'єднання, тобто за визначення, чи інша сторона все ще існує **і** доступна, навіть якщо між вами довго не було обміну даними.

У TCP немає вбудованого механізму "автоматичного виявлення розриву зв'язку", якщо:

- ніхто не надсилає дані;

- підключення “зависло” через обрив мережі, падіння **Wi-Fi**, вимкнений ноутбук клієнта, завислий роутер тощо.

З погляду операційної системи, сокет усе ще **“підключений”**, бо формального **`FIN`** чи **`RST`** не прийшло.

Якщо встановити цю опцію, то TCP-стек ОС **періодично надсилає службові “keepalive-пакети”** на іншу сторону, якщо протягом тривалого часу **не було жодного трафіку**. Якщо інша сторона не відповідає - ядро **вважає з’єднання “мертвим”** і повідомляє програму через помилку (**`ETIMEDOUT`**, **`ECONNRESET`** тощо).

Налаштування інтервалів:

Функція **setsockopt()** дозволяє налаштувати інтервал перевірки:

```
int idle = 60;          // через 60 сек почати перевірку після останньої активності
int intvl = 10;         // інтервал між перевірками 10 сек
int cnt = 3;            // 3 спроби
setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPIDLE, &idle, sizeof(idle));
setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPINTVL, &intvl, sizeof(intvl));
setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPCNT, &cnt, sizeof(cnt));
```

У лабораторній роботі, щоб система дозволила створювати сокет при наступному запуску програми слід налаштувати сокет (мінімальне налаштування):

```
int opt = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

Це дозволить використовувати сокет повторно, не чекаючи, пока операційна система “забуде” про нього.

Сервер

Сервер – це програма, або інша якась суть, яка надає у вільне користування свої ресурси.

Кліент – це програма, або інша якась суть, яка відправляє серверу запит на отримання тої чи іншої інформації, а отпримавши відповідь у зручній формі повідомляє користувачу.

Створивши сокет необхідно “зв’язати” його з мережевим інтерфейсом хосту у якому він був створений. Для цього існує функція ядра операційної системи **bind()**. Інформація о мережевом інтерфейсі записується у структуру типу **sockaddr_in** (для домену **AF_INET**):

```
#include <arpa/inet.h>
struct sockaddr_in {
     sa_family_t sin_family;          // Domain
     in_port_t sin_port;              // Port
     struct in_addr_t sin_addr;       // Interface
     char sin_zero[8];                // Заповнювач до розміру як sockaddr
};

struct in_addr {
     uint32_t s_addr;                 // IPv4-адреса у вигляді 32-бітного числа
}
```

Наприклад:

```
in_port_t port = htons(12345);              // Платформонезалежне значення порту
in_addr_t addr = inet_addr(“localhost”); // або INADDR_ANY
struct sockadd_in s_addr = { AF_INET, port, addr };
```

Після цього можемо викликати функцію **bind()**:

```
#include <sys/socket.h>
#include <arpa/inet.h>
int bind(socket, (sockaddr*) &s_addr, sizeof(s_addrt));
```

Наприклад:

```
struct sockadd_in s_addr = { AF_INET, port, addr };
if (bind(sid, (struct sockaddr*) &s_addr, sizeof(s_addr)) < 0) {
```

```
        printf("Bind error\n");
        exit(1);
}
```

Системний виклик **bind()** приймає вказівник на структуру з параметрами мережевого інтерфейсу типу **sockaddr**, тому, при передачі адреси на структуру **sockaddr_in** необхідно явне перетворення типу (наприклад, для комунікаційного домену **AF_UNIX** така структура має тип **sockaddr_un**).

Зауваження: З*аголовочний файл* ***<sys/socket.h>*** *дозволяється явно не підключати до програми, оскільки підключення файлу* ***<arpa/inet.h>*** *автоматично підключає і файл* ***<sys/socket.h>***.

Сервер чекає запист від кліенту, обробляє його та надсилає відповідь. Таким чином нам необхідно перевести сокет у режим очикування запиту на підключення. Для цього необхідно створити чергу підключень:

```
#include <sys/socket.h>
int listen(socket, length_queue);
```

Наприклад:

```
if (listen(sid, 0) < 0) {
        printf("Listen error\n");
        exit(1);
}
```

Параметр **length_queue** вказує на довжину черги, значення **0** означає максимально можливу.

Теперь все готово для прослухування запитів на підключення. За це відповідає системний виклик **accept()**, який блокує викликаючий його процес до отримання запиту на підключення:

```
#include <sys/socket.h>
#include <arpa/inet.h>
struct sockadd_in s_addr;
socklen_t s_leng = sizeof(s_addr);
int accept(socket, (struct sockaddr*) &s_addr, &s_leng);
```

У структуру **s_addr** функція **accept()** запише параметри сокету кліента, з якого надіслано запит на підключення. Наприклад, отримати **ip** адресу кліента:

```
#include <arpa/inet.h>
char* ip = inet_ntoa(s_addr.sin_addr);
```

або порт його сокету, який надіслав запит:

```
#include <arpa/inet.h>
in_port_t port = htons(s_addr.sin_port);
```

У якості результату системний виклик повертає файловий дескриптор сокету, який був створений функцією **accept()** через який відбудеться передача даних між клієнтлм та сервером.

Щоб сервер міг прослуховувати багато підключень (по черзі) необхідно функцію **accept()** викликати у циклі, а обробку поточного запиту на підключення здіснювати як омога швидше.

Наприклад:

```
while (true) {
      int cid = accept(socket, (struct sockaddr*) &s_addr, &s_leng);
      // Обробка запиту
      // ..
      close(cid);  // Закрити з'єднання
}
```

Запис у сокет та читання з нього

Після підключення кліент може надіслати повідомлення до серверу. Щоб прочитати його, використовують системній виклик **read()**, а для відправки відповіді – системний виклик **write()**. У якості файлового дескриптору для функцій **read()** та **write()** слід вказати дескриптор кліент-сокету, яких повернув метод **accept()**.

```
#include <unistd.h>
size_t read(int fd, char* buf, int length_buf);
   повертає кількість прочитаних байтів, або -1, якщо виникла помилка

size_t write(int fd, char* buf, int length_buf);
   повертає кількість записаних байтів, або -1, якщо виникла помилка
```

Наприклад:

```
…
char buf[1024] = { 0 };
   // Створили буфер
int countr = read(cid, buf, sizeof(buf));
   // Читаємо із сокету у буфер не більше ніж sizeof(buf) символів
printf(“Прочитано %d байт, s = %s\n”, countr, buf);
 int countw = write(cid, buf, sizeof(buf));
   // Записуємо (відправляємо) у сокет отриманий рядок (echo-server)
printf(“Записано %d байт, s = %s\n”, countw, buf);
…
```

При читанні та відправці даних можливо, що зв’язок буде перервано, у цьому випадку функції **read()** та/або **write()** надсилає власному процесу сигнал **SIGPIPE**. Якщо цей сигнал не перехопити, то стандартна реакція процесу на отримання цього сигналу – це аварійна зупинка програми. Програміст має можливість перевизначити диспозіцію вказанного сигналу.

Простий сервер

Нище приводиться повний код програми по створенню простого серверу, який дозволяє одночасно підключитись тільки одному користувачу. Користувач підключившись, надсилає повідомлення, на яке сервер відповідає словом ‘**YES**’ та чекає наступного повідомлення. Для завершення сеансу роботи користувач надсилае повідомлення ‘**exit**’. Отримавши таке повідомлення сервер завершує сеанс роботи з клієнтом, та переходить до прослухування наступного запиту на підключення.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

int main() {
  signal(SIGPIPE, SIG_IGN);
  sa_family_t domain = AF_INET;
  int sid;
  if ((sid = socket(domain, SOCK_STREAM, 0)) < 0) {
    printf("Socket error\n");
    exit(1);
  }
  int opt = 1;
  setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
  in_port_t port = htons(12345);
  in_addr_t addr = INADDR_ANY;     // Дозволити прослуховувати усі інтерфейси
  struct sockaddr_in s_addr = { domain, port, addr };
  if (bind(sid, (struct sockaddr*) &s_addr, sizeof(s_addr)) < 0) {
    printf("Bind error\n");
    exit(1);
  }
  listen(sid, 0);      // Максимально можлива черга
  int cid;
  struct sockaddr_in c_addr;
  socklen_t len = sizeof(c_addr);
  while ((cid = accept(sid, (sockaddr*) &c_addr, &len)) != -1) {
```

```
    char c[260];
    do {
      memset(c, 0, 260);
      if (read(cid, c, 260) > 0) {
        printf("%s\n", c);
        write(cid, "YES\n", 4);
      } else {
        strcpy(c, "exit");
      }
    } while (strcmp(c, "exit") != 0);
    printf("DISCONNECT\n");
    close(cid);
  }
  exit(0);
}
```

Клиент

Програма-кліент після створення сокету повина спробувати підключитися до серверу. Для цього викликається функція ядра операційної системи **connect()**, і у разі успіху з’єднання буде створено і можлива передача даних від кліента до серверу та навпаки

```
#include <arpa/inet.h>
in_port_t port = htons(12345);            /       / Порт прослуховування сервером
in_addr_t addr = inet_addr(“172.16.8.2”);       // Адреса серверу
struct sockadd_in s_addr = { AF_INET, port, addr };
```

Після цього можемо викликати функцію **connect()**:

```
#include <sys/socket.h>
#include <arpa/inet.h>
int connect(socket, (sockaddr*) &s_addr, sizeof(s_addrt));
```

Наприклад:

```
struct sockadd_in s_addr = { AF_INET, port, addr };
if (connect(sid, (struct sockaddr*) &s_addr, sizeof(s_addr)) < 0) {
     printf(“Connect error\n”);
     exit(1);
}
```

Якщо з’єднення вдалося, то можно надіслати данні у сокет та читати з нього. Для цього використовують стандартні функції ядра операційної системи **write()** та **read()**.

```
#include <unistd.h>
size_t write(socket, buffer, length_buffer);
size_t read(socket, buffer, length_buffer);
   Функції повертають кількість відправлених/отриманих байтів
```

Наприклад:

```
char buf[1024];
read(sid, buf, strlen(buf));
write(sid, buf, strlen(buf));
```

Додаток 1

Список студентів по курсу «Операційні системи» для визнання номеру індівідуального завдання.

1	Шевченко Богдан Миколайович	ylua5555@gmail.com
2	Адамов Ярослав Родіонович	larchistrike@gmail.com
3	Астапенко Кирило Тарасович	kirill.ast98@gmail.com
4	Башлай Владислав Віталійович	vladbashlay123@gmail.com
5	Бурмагін Нікіта Сергійович	nnburmagin@gmail.com
6	Дереча Олексій Іванович	lollexa215@gmail.com
7	Доценко Денис Олексійович	denis.dotsenko.al@gmail.com
8	Дячко Діана Євгенівна	dianadiachko@gmail.com
9	Євстігнєєв Ростислав Денисович	yevstihnieiev.uni@gmail.com
10	Згурський Данило Олександрович	danilzgurs@gmail.com
11	Здор Віталій Віталійович	azot587@gmail.com
12	Іванченко Софія Олександрівна	sonyaivanchenko1@gmail.com
13	Кабанов Артем Іванович	artemkabanov108@gmail.com
14	Каптюх Костянтин Денисович	ravolut1onr@gmail.com
15	Караваєв Олександр Олександрович	koshak327@gmail.com
16	Колєснік Максим Олександрович	maksimkolesnik99@gmail.com
17	Крутько Дмитро Геннадійович	durkomom735@gmail.com
18	Кузін Іван Олексійович	ivankuzin3049@gmail.com
19	Лукіна Дарія Дмитрівна	lukinadariia@gmail.com
20	Матяш Ярослав Олександрович	matyhya23@gmail.com
21	Мірошниченко Богдан Володимирович	bogdanmirosnicenko@gmail.com
22	Могилін Владислав Олександрович	mogilinvlad@gmail.com
23	Мороко Ярослав Владиславович	legionemptys@gmail.com
24	Петрик Денис Сергійович	denispetrik714@gmail.com
25	Пригарін Богдан Сергійович	pryharin.bohdan@gmail.com
26	Сандак Поліна Андріївна	charasailar@gmail.com
27	Сапа Олександр Олександрович	sahasapa2000@gmail.com
28	Сапа Сергій Олександрович	serchie228@gmail.com
29	Тарасенко Нікіта Сергійович	tarasenkonikita27@gmail.com
30	Темченко Станіслав Андрійович	temchenkosv8@gmail.com
31	Хоренженко Іван Сергійович	ivankhorenzhenko22@gmail.com
32	Черевичний Ярослав Сергійович	yarik16122006@gmail.com
33	Черненко Едуард Кирилович	edikchernenko133@gmail.com
34	Шарій Юрій Андрійович	shariyya777@gmail.com