Семестр 4 (2017), TCP/IP «Эхо» сервер, клиент

Сетевое приложение состоит из двух программ – клиент и сервер.

Клиент (общая структура программы)

Программа клиента имеет следующую структуру. В функции `main` создается сетевой сокет и устанавливается соединение с сервером. Вызывается функция `writeRead`, которая отсылает текстовое сообщение на сервер, принимает ответ от сервера и печатает этот ответ в стандартный поток вывода. После этого в функции `main` закрывается сетевое соединение.

В качестве текстового сообщения используется либо первый аргумент командной строки, либо строка `This is a test message`.

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#include <string.h>
#include <stdio.h>

static void writeRead(int fd, const char *txt);

int main(int argc, char *argv[]) {
    ...
    writeRead(fd,  argc > 1
                ? argv[1]
                : "This is a test message");
    ...
}

static void writeRead(int fd, const char *txt) {
    ...
}
```

Последовательно разберем функции `main` и `writeRead`.

Клиент (функция `main`)

В начале функции `main` определяются четыре локальные переменные. Через переменную `host` задается строковое представление IP-адреса хоста, на котором запущен сервер, а через переменную `port` задается номер порта, который «слушает» сервер. Переменная `addr` будет содержать адрес сокета, а `fd` – его файловый дескриптор.

```
1  const char         *host = "127.0.0.1";
2  uint16_t            port = 8000;
3  struct sockaddr_in addr;
4  int                fd;
```

Будем работать с сокетами, использующими способ адресации семейства протоколов IPv4. Данный способ адресации задается константой `AF_INET`. Адрес сокета будет состоять из IP-адреса и номера порта. IP-адрес представляет собой последовательность из четырех байтов. Он может быть задан как беззнаковое четырех байтовое целое число, а также может быть задан в текстовом виде. Текстовое представление IP-адреса состоит из четырех чисел в диапазоне от 0 до 255, записанных через точку. Номером порта может быть целое число в диапазоне от 1 до 65535.

В строке 5 с помощью вызова стандартной функции `memset` «обнуляется» область памяти (каждой ее ячейке присваивается значение 0), выделенная под хранение переменной `addr`. В строке 6 полю `sin_family` переменной `addr` присваивается значение `AF_INET`. Тем самым устанавливается выбранный способ адресации. В строке 7 в поле `sin_port` переменной `addr` сохраняется номер порта. При этом сохраняемое число должно быть закодированно с использованием сетевого порядка байт. Для выполнения в случае необходимости соответствующей перекодировки используется функция `htons`. В строках 8-10 с помощью вызова функции `inet_pton` текстовое представление IP-адреса преобразуется в числовое представление, которое в свою очередь сохраняется в поле `sin_addr` переменной `addr`. В случае неудачи данная функция вернет значение меньшее 1.

```
5  memset(&addr, 0, sizeof addr);
6  addr.sin_family = AF_INET;
7  addr.sin_port   = htons(port);
8  if(inet_pton(AF_INET,
9               host,
10              &addr.sin_addr) < 1) {
11     fprintf(stderr, "Wrong host value\n",);
12     return -1;
13 }
```

Сокет создается с помощью системного вызова `socket()` (строка 14). При его вызове передаются три входных параметра. Первый параметр определяет способ адресации. Второй параметр задает тип коммуникации. Константа `SOCK_STREAM` соответствует потоковой коммуникации. Сокет данного типа представляет собой двунаправленный канал, доступный с обоих концов как для записи (например, с помощью системного вызова `write()`), так и для чтения (например, с помощью системного вызова `read()`). Третий параметр задает протокол. Константа `0` соответствует единственно возможному для данной комбинации способ адресации и типа коммуникации протоколу TCP.

В случае успеха системный вызов `socket()` вернет неотрицательное целое число, представляющее собой файловый дескриптор, привязанный к созданному сокету. В случае неудачи будет возвращено значение `-1`.

```
14 fd = socket(AF_INET, SOCK_STREAM, 0);
15 if(fd == -1) {
16     fprintf(stderr, "Can't create socket\n");
17     return -1;
18 }
```

Для установления соединения с сервером используется системный вызов `connect()` (строки 19-21). При его вызове передаются три входных параметра. Первый параметр – это файловый дескриптор сокета, созданного для коммуникации с сервером. Второй параметр – это указатель на структуру, содержащую адрес сервера. Третий параметр представляет собой размер в байтах этой структуры.

```
19  if(connect(fd,
20             (struct sockaddr *)&addr,
21             sizeof addr) == -1) {
22      fprintf(stderr, "Can't connect\n");
23      if(close(fd))
24          fprintf(stderr, "Can't close\n");
25      return -1;
26  }
```

В случае успешной установки соединения вызывается функция `writeRead` для отправки текстового сообщения на сервер и приема ответа от сервера.

```
27  writeRead(fd,  argc > 1
28                ? argv[1]
29                : "This is a test message");
```

С помощью системного вызова `shutdown()` (строка 30) завершается работа с сокетом. При его вызове передаются два входных параметра – файловый дескриптор сокета и числовой параметр, определяющий какой вид работы с сокетом завершается (0 – сокет закрывается на чтение, 1 – сокет закрывается на запись и 2 – сокет закрывается как на чтение так и назапись).

```
30  if(shutdown(fd, 2) == -1)
31      fprintf(stderr, "Can't shutdown\n");
```

После завершения работы с сокетом необходимо освободить ассоциированный с ним файловый дескриптор. Для этого используется системный вызов `close()` (строка 32).

```
32  if(close(fd)) {
33      fprintf(stderr, "Can't close\n");
34      return -1;
35  }
36
37  return 0;
```

Клиент (функция `writeRead`)

Функция `writeRead` в качестве входных параметров получает файловый дескриптор `fd` сокета, созданного для коммуникации с сервером, а также текстовую строку `txt`, которую необходимо отправить на сервер.

В начале тела функции объявляются две локальные переменные `len` и `buf`. Значением переменной `len` является четырех байтовое беззнаковое целое число. В строке 4 в эту переменную сохраняется длина последовательности байтов, которая содержит символы строки `txt`, включая завершающий нулевой символ-признак конца строки. Переменная `buf` представляет собой массив байтов размера 1024, в который будет сохранятся полученный с сервера ответ.

```
1  uint32_t len;
2  char     buf[1024];
3
4  len = strlen(txt) + 1;
```

В строке 5 с помощью системного вызова `write()` осуществляется попытка отправить на сервер (записать в сокет) последовательность из четырех байтов, в которых закодирована длина строки `txt`. Если не удалось за один раз целиком оправить эту последовательность байтов, то это трактуется как ошибка.

Аналогично в строке 10 осуществляется попытка отправить за один раз последовательность байтов, содержащую символы строки `txt`.

```
5  if(write(fd, &len, sizeof len) != sizeof len) {
6      fprintf(stderr, "Can't write length\n");
7      return;
8  }
9
10 if(write(fd, txt, len) != (ssize_t)len) {
11     fprintf(stderr, "Can't write text\n");
12     return;
13 }
```

В строке 14 осуществляется попытка прочитать последовательность из четырех байтов, в которых закодирована длина текстового сообщения, которое сервер планирует отправить в качестве ответа клиенту. Длина последовательности сохраняется в переменную `len`. Если не удалось за один раз целиком прочитать четырех байтовую последовательность, то это трактуется как ошибка.

В строке 19 проверяется возможность записать сообщение, которое планирует отправить сервер, в массив `buf`.

```
14 if(read(fd, &len, sizeof len) != sizeof len) {
15     fprintf(stderr, "Can't read length\n");
16     return;
17 }
18
19 if(len > sizeof buf) {
20     fprintf(stderr, "Too big message\n");
21     return;
22 }
```

В строке 23 осуществляется попытка за один раз целиком прочитать последовательность байтов, содержащую текстовый ответ сервера, и записать ее в массив `buf`.

В строке 28 получееный от сервера ответ печатается в стандартный поток вывода.

```
23 if(read(fd, buf, len) != (ssize_t)len) {
24     fprintf(stderr, "Can't read text\n");
25     return;
26 }
27
28 puts(buf);
```

Сервер (общая структура программы)

Программа сервера имеет следующую структуру. В функции `main` создается слушаюший сокет для приема входящих соединений. Вызывается функция `loop`, которая содержит «бесконечный» цикл. В рамках очередной итерации цикла принимается входящее соединение от клиента. Через это соединение сервер получает отпраленное клиентом тектовое сообщение. Добавляет этому сообщению префикс `Echo:` и отправляет модифицированное таким образом текстовое сообщение обратно клиенту. После этого закрывает соединение.

Последовательно разберем функции `main` и `loop`.

```
#include <arpa/inet.h>
#include <sys/types.h>
```

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

#include <inttypes.h>
#include <string.h>
#include <stdio.h>

static void loop(int ld);

int main(void) {
    ...
    loop(ld);
    ...
}

static void loop(int ld) {
    ...
}
```

Сервер (функция `main`)

В начале функции `main` определяется пять локальных переменных. Переменные `host`, `port`, `addr` имеют то же назначение, что и в программе клиента. Переменная `ld` будет содержать файловый дескриптор слушающего сокета. Переменная `on` будет использоваться для установления на этом сокете опции `SO_REUSEADDR`.

```
1  const char        *host = "127.0.0.1";
2  uint16_t           port = 8000;
3  struct sockaddr_in addr;
4  int                ld;
5  int                on;
```

В строках 6 – 20 формируется в качестве значения переменной `addr` адрес сокета, а также с помощью системного вызова `socket()` создается сам сокет.

```
6  memset(&addr, 0, sizeof addr);
7  addr.sin_family = AF_INET;
8  addr.sin_port   = htons(port);
9  if(inet_pton(AF_INET,
10              host,
11              &addr.sin_addr) < 1) {
12     fprintf(stderr, "Wrong host value\n");
13     return -1;
14 }
15
16 ld = socket(AF_INET, SOCK_STREAM, 0);
17 if(ld == -1) {
18     fprintf(stderr, "Can't create socket\n");
19     return -1;
20 }
```

В некоторых случаях после завершения выполнения программы сервера ядро операционной системы может продолжать какое-то время считать использовавшийся сетевой адрес занятым. Поэтому какое-то время запустить сервер с тем же номером порта не удастся. Чтобы изменить подобное поведение ядра операционной системы необходимо на будующем (до выполнения системного вызова `bind()`) слушающем сокете выставить опцию `SO_REUSEADDR`. Это осуществляется с помощью системного вызова `setsockopt()` (строки 21 – 29).

```
21 on = 1;
22 if(setsockopt(ld,
23                SOL_SOCKET,
24                SO_REUSEADDR,
25                &on,
26                sizeof on) == -1) {
27     fprintf(stderr, "Can't setsockopt\n");
28     return -1;
29 }
```

С помощью системного вызова `bind()` (строки 30 – 35) сокет связывается с сетевым адресом. При его вызове передаются три входных параметра. Первый параметр – это файловый дескриптор слушающего сокета, созданного для приема сетевых соединений. Второй параметр – это указатель на структуру, содержащую адрес, с которым связывается этот сокет. Третий параметр представляет собой размер в байтах этой структуры. В случае успеха возвращается значение 0, а в случае неудачи – значение -1.

```
30 if(bind(ld,
31          (struct sockaddr *)&addr,
32          sizeof addr) == -1) {
33     fprintf(stderr, "Can't bind\n");
34     return -1;
35 }
```

После создания сокета и привязки к нему сетевого адреса сокет должен быть переведен в состояние ожидания запросов на соединения. С этой целью используется системный вызов `listen()` (строки 36 – 39). Первым входным параметром является файловый дескриптор сокета, который в случае успеха системного вызова становится слушающим. Второй параметр представляет собой максимальную длину очереди запросов на соединение. В случае успеха возвращается значение 0, а в случае неудачи – значение -1.

```
36 if(listen(ld, 5) == -1) {
37     fprintf(stderr, "Can't listen\n");
38     return -1;
39 }
```

Далее, запускается функция `loop`, которая содержит «бесконечный» цикл. В рамках каждой итерации цикла обрабатывается запрос на соединение. Обработка включает установку соединения. Прием текстового сообщения от клиента. Подготовку ответа и его отправку клиенту. Закрытие соединения.

```
40 loop(ld);
```

После завершения функции `loop` освобождается файловый дескриптор, ассоциированный со слушающим сокетом.

```
41 if(close(ld)) {
42     fprintf(stderr, "Can't close\n");
43     return -1;
44 }
45
46 puts("Done.");
47 return 0;
```

Сервер (функция `loop`)

В начале тела функции `loop` объявляются пять локальных переменных. Переменные `addr`, `addrlen` и `fd` используются для выполнения системного вызова `accept()`. Переменная `len` будет использоваться для хранения длины полученного от клиента текстового сообщения и для хранения длины ответа, который сервер будет переда-

вать клиенту. Сами эти текстовые сообщения будут сохраняться и формироваться в массиве байтов `buf`.

Строка 8 содержит вызов стандартной функции `strcpy`, с помощью которой в начало массива `buf` записываются символы `Echo:` .

```
1  struct sockaddr_in addr;
2  socklen_t          addrlen;
3  int                fd;
4
5  uint32_t           len;
6  char               buf[1024];
7
8  strcpy(buf, "Echo: ");
```

Запросы на соединение от клиента обрабатываются в цикле (строки 9 – 52). В рамках каждой итерации цикла осуществляется следующая последовательность действий.

С помощью стандартной функции `memset` (строка 10) «зануляется» область памяти, выделенная под хранение переменной `addr`. В переменную `addrlen` сохраняется (строка 11) размер этой области памяти.

Строки 13 – 15 содержат системный вызов `accept()`. Системный вызов получает на вход три входных параметра. Первым параметром является файловый дескриптор слушающего сокета. Втором параметром является указатель на область памяти, куда будет записана структура данных, содержащая адрес клиента, выставившего запрос на установку соединения. Третьим параметром является указатель на переменную, содержащую размер этой области памяти. В случае успешного выполнения системного вызова по адресу этой переменной будет записан фактический размер структуры данных с адресом клиента.

В случае успеха возвращается неотрицательное целое число – файловый дескриптор, связанный с сокетом, через который будет осуществляться взаимодействие с клиентом. В случае ошибки системный вызов вернет значение -1.

```
 9  for(;;) {
10      memset(&addr, 0, sizeof addr);
11      addrlen = sizeof addr;
12
13      fd = accept(ld,
14                  (struct sockaddr *)&addr,
15                  &addrlen);
16      if(fd == -1) {
17          fprintf(stderr, "Can't accept\n");
18          continue;
19      }
```

В строке 20 осуществляется попытка прочитать за один раз длину текстового сообщения клиента и записать ее по адресу переменной `len`. В случае неудачи осуществляется переход (строка 22) к закрытию соединения (строка 45) и освобождению файлового дескриптора сокета (строка 49).

Для перехода используется оператор `goto`. При написании программы на языке Си++ рекомендуется заменить использование оператора `goto` на конструкцию `try-catch`.

```
20      if(read(fd, &len, sizeof len) != sizeof len) {
21          fprintf(stderr, "Can't read length\n");
22          goto end;
23      }
24
25      printf("Conn: %d Len: %"PRIu32"\n", fd, len);
```

В строке 26 проверяется, что текстовое сообщение от клиента в случае его получения может быть сохранено в массив `buf`.

```
26      if(len + 6 > sizeof buf) {
27          fprintf(stderr, "Too big message\n");
28          goto end;
29      }
```

В строке 30 осуществляется попытка за один раз прочитать текстовое сообщение от клиента и записать его в область памяти по адресу `buf + 6`.

```
30      if(read(fd, buf + 6, len) != (ssize_t)len) {
31          fprintf(stderr, "Can't reade text\n");
32          goto end;
33      }
34
35      printf("Conn: %d Body: %s\n", fd, buf + 6);
```

В строке 37 осуществляется попытка за один раз отправить клиенту длину ответного сообщения.

```
36      len += 6;
37      if(write(fd, &len, sizeof len) != sizeof len) {
38          fprintf(stderr, "Can't write length\n");
39          goto end;
40      }
```

В строке 41 осуществляется попытка за один раз отправить клиенту ответное сообщение.

```
41      if(write(fd, buf, len) != (ssize_t)len) {
42          fprintf(stderr, "Can't write text\n");
43      }
```

Как уже было отмечено ранее, в строке 45 осуществляется системный вызов `shutdown()`, с помощью которого звершается работа с соединением. Сокет закрывается на чтение и на запись.

В строке 49 осуществляется системный вызов `close()`, освобождающий файловый дескритор, ассоцированный с закрытым соединением.

```
44  end:
45      if(shutdown(fd, 2) == -1) {
46          fprintf(stderr, "Can't shutdown\n");
47      }
48
49      if(close(fd)) {
50          fprintf(stderr, "Can't close\n");
51      }
52  }
```

Сигналы

Программа сервера содержит функцию `loop`, которая в свою очередь содержит «бесконечный цикл», в рамках которого обрабатываются входящие соединения от клиентов. В начале этой функции присутствует системный вызов `accept()`, который блокирует выполнение программы до появления очередного запроса на установку соединения. Как в этой ситуации осуществить корректное завершение программы сервера?

Прекратить программу можно нажав комбинацию клавиш `Ctrl+C`. Программе будет послан сигнал `SIGINT`. По умолчанию обработка этого сигнала заключается в немедленном принудительном

завершении программы. Подобная ситуация не может рассматриваться как удовлетворительная. Например, при таком прерывании не будут выполняться строки 41 – 47 программы сервера, в которых корректно освобождается файловый дескриптор слушающего сокета.

Однако, можно задать собственный обработчик сигнала. В следующем фрагменте программы определяются две функции. Функция `handler`, которая ничего «не делает», будет использоваться в качестве нового обработчика сигналов. Функция `signalIgnoring` с помощью системного вызова `sigaction()` устанавливает функцию `handler` в качестве обработчика сигналов `SIGINT` и `SIGPIPE`.

```
#include <signal.h>
...
static void handler(int signo) {
    (void)signo;
}

static int signalIgnoring(void) {
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags   = 0;
    sigemptyset(&act.sa_mask);

    if(sigaction(SIGINT, &act, 0) == -1) {
        fprintf(stderr, "Can't SIGINT ignoring\n");
        return -1;
    }

    if(sigaction(SIGPIPE, &act, 0) == -1) {
        fprintf(stderr, "Can't SIGPIPE ignoring\n");
        return -1;
    }

    return 0;
}
```

Вызов функции `signalIgnoring` может быть помещен в функцию `main` программы сервера перед строкой 6. Теперь нажатии клавиш `Ctrl+C` управление будет передаваться в обработчик `handler`, а за тем возвращаться назад в то место программы, где было осуществлено прерывание. Если в этом месте осуществлялся системный вызов `accept()`, то он завершится с ошибкой (вернет значение -1). При этом глобальной переменной `errno` будет присвоено значение `EINTR`.

Строки 16 – 19 функции `loop` следует заменить следующим образом. После обработки прерывания функция `loop` будет завершать свое выполнение. Будет осуществляться выход из бесконечного цикла.

```
#include <errno.h>
...
static void loop(int ld) {
    ...
    if(fd == -1) {
        if(errno == EINTR)
            return;
        else {
            fprintf(stderr, "Can't accept\n");
            continue;
        }
    }
    ...
}
```