Программирование Статьи Сеть

Полное руководство по сетевому программированию для разработчиков игр. Часть 1 (скучная). (2 стр)

socket() - окно в мир

"Петр Первый прорубил окно в Европу вместо того,
чтобы просто пройти через дверь"

Перед тем, как использовать код сокетов, надо подключить соответствующие библиотеки и вставить соответствующие хэдеры (эээ... меня всегда интересовало, почему люди говорят "хИдеры" вместо более правильного "хЭдеры", ведь header (заголовок) происходит от слова head (голова))... Как подключать и вставлять, надеюсь объяснять не надо, а если и надо, то, значит тебе, дорогой читатель, надо было начинать изучение программирования с чего-нибудь полегче...

Код для инициализации и создания сокетов в системах *nix и Windows выглядит по-разному, это обусловлено тем, что разработчики Windows вынесли код сетевой подсистемы в отдельную библиотеку, поэтому программистам под Windows придется провести дополнительную инициализацию, чтобы сетевая подсистема дала возможность работать с сокетами.

Создание сокета выглядит так:

```
// Listing 1.01 win
// объявление функции выглядит так:
// SOCKET socket (int pf, int type, int protocol);

#pragma comment (lib, "ws2_32.lib");    // ищем нужную библиотеку
#include <winsock2.h>        // winsock2.h: typedef u_int SOCKET

  WORD wVersion;          // запрашиваемая версия winsock-интерфейса
  WSADATA wsaData;        // сюда записываются данные о сокете
  wVersion = MAKEWORD (2, 0);      // задаем версию winsock
  SOCKET sd;          // наш дескриптор сокета

  int wsaInitError = WSAStartup (wVersion, &wsaData);    // инициализируем winsock
  if (wsaInitError != 0)
    // говорим пользователю, что возникли проблемы и выходим
    exit (1);
  else
    // если инициализация прошла успешно, то пора создавать сокет
    sd = socket (PF_INET, SOCK_DGRAM, 0);


// Listing 1.01 nix
// объявление выглядит так:
// int socket (int domain, int type, int protocol);

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

  int sd;            // наш дескриптор

  // здесь нет инициализации, потому что socket() - это системный вызов в *nix
  sd = socket (PF_INET, SOCK_DGRAM, 0);
```

В принципе, из комментариев все ясно, но все же рассмотрим socket() по порядку:

domain (domain == pf == protocol family) - это предопределенная константа, от которой зависит стек протоколов (домен) с которым мы будем работать. Здесь надо сделать небольшую ремарку... Компьютеры общаются между собой на "сетевом языке". Таких сетевых языков существует целое множество, но вряд ли два компьютера поймут друг друга, если они будут разговаривать на разных языках ("твоя моя не понимать, геолога плохой, чукча хороший"). "Правила грамматики" для каждого из языков определяются протоколом. Например, протокол IP (Internet Protocol), протокол TCP (Transmission Control Protocol - протокол контроля за передачей данных), UDP (User Datagram Protocol - протокол пользовательских дэйтаграмм) и т.д. На обоих концах соединения компьютеры должны выбрать один и тот же протокол, чтобы начать общение. Так вот domain как раз и определяет уровень протоколов, с которыми наше приложение будет работать... Сразу оговоримся, что общепризнанным стандартом являются коммуникации по Internet, где заслуженно царствует стек (набор) протоколов под названием TCP/IP. Его-то мы и будем юзать вовсю. Однако здесь есть один тонкий момент. В вызове socket() надо использовать набор констант PF_XXXX, а не AF_XXXX (AF_ == Address Family), как рекомендуют многие руководства по сетевому программированию. В настоящий момент константы PF_XXXX и AF_XXXX взаимозаменяемы, но эта ситуация может измениться в будущем, так что следует отнестись к этому внимательно. Обо всех вариантах констант PF_XXXX (PF_ == Protocol Family - адресное пространство) можно узнать из документации или просмотрев хэдерные файлы (Windows - winsock2.h + MSDN; *nix - man 2 socket + sys/socket.h). На данный момент мы будем использовать PF_INET (семейство адресов Internet).

type - это тип протокола, принадлежащего выбранному семейству адресов, который мы выбираем для передачи данных. Стопроцентной переносимостью хвалятся только два варианта: SOCK_STREAM (для протокола TCP) и SOCK_DGRAM (для UDP). О поддерживаемых вариантах надо смотреть документацию для каждой конкретной ОС (и вообще!? Когда же вы научитесь читать доки?) :)

protocol - данное число определяет, какой из протоколов мы будем использовать. В зависимости от выбранного семейства адресов и типа протокола, данный аргумент будет принимать разные значения... Спешу тебя обрадовать, нам об этом числе в данный момент думать не надо, потому что семейство адресов и тип протокола мы уже выбрали, а для связки PF_INET + SOCK_STREAM/SOCK_DGRAM параметр protocol всегда равен нулю. :) Т.е. это число нужно только тогда когда мы углубляемся в реализацию сокетов или создаем свой собственный протокол. Мы пока не будем этим заниматься (нам еще рано), поэтому для нас оно равно нулю, и точка.

Итак, мы заказали нашей ОС создать сокет с такими-то параметрами. Вызов socket() возвращает либо дескриптор сокета, либо число, сигнализирующее об ошибке. Дескриптор сокета - это обычное целое число, которое идентифицирует ресурсы, с которыми работает сокет. Его непосредственное значение нас мало должно заботить, нам оно нужно, чтоб передавать его как параметр для других вызовов. Вообще говоря, в системе может работать куча разных приложений, которые в свою очередь, могут иметь очень много открытых дескрипторов сокетов. Вот как раз для того, чтобы отличать один сокет от другого, нам и нужен его дескриптор. Дескриптор сокета подобен дескриптору файла, более того, мы даже можем привести его к файловому типу и использовать стандартные вызовы read()/write() чтобы принимать и отсылать данные, но об этом чуть позже...

Итак, при вызове socket() мы делаем следующее:

```
#ifdef WINDOWS
SOCKET sd = socket (PF_INET, SOCK_DGRAM, 0);   // код для систем windows
#else
int sd = socket (PF_INET, SOCK_DGRAM, 0);    // для систем *nix
#endif
```

Самое время поговорить о непредвиденных обстоятельствах, то есть об ошибках. И тут опять стоит сделать небольшое лирическое отступление. По стандарту родных для сокетов систем *nix код ошибок в подобных вызовах записывается в глобальную библиотечную переменную errno. Однако разработчики winsock2 не совсем согласны с подобной политикой, поэтому в Windows-системах для получения кода последней зафиксированной ошибки надо использовать int WSAGetLastError(void). Но! Из объявления функции socket() видно, что в случае с Windows-системами она может возвращать только неотрицательные числа, тогда как классический BSD socket() возвращает любые из диапазона int. Для сигнализации об ошибке winsock socket() возвращает 0, а BSD socket() возвращает отрицательное число -1. Получается, что дескриптор, который равен нулю в *nix системах, считается легальным, а в Windows - нет. Здесь стоит быть крайне внимательным к обработке ошибок. В общем случае код для обработки ошибок в вызове socket() может выглядеть так:

```
// Listing 1.02 win

sd = socket (PF_INET, SOCK_DGRAM, 0);   // создаем сокет

if (sd == INVALID_SOCKET)   // winsock2.h: #define INVALID_SOCKET (SOCKET)(~0)
{
  int wsError = WSAGetLastError();
  switch (wsError)
  {
    case WSANOTINITIALISED:   // здесь обрабатываем ошибку
      break;
    case WSAENETDOWN:      // здесь обрабатываем ошибку
      break;
    case WSAEAFNOSUPPORT:    // здесь обрабатываем ошибку
      break;
    case WSAEINPROGRESS:    // здесь обрабатываем ошибку
      break;
    case WSAEMFILE:      // здесь обрабатываем ошибку
      break;
    case WSAENOBUFS:      // здесь обрабатываем ошибку
      break;
    case WSAEPROTONOSUPPORT:  // здесь обрабатываем ошибку
      break;
    case WSAEPROTOTYPE:    // здесь обрабатываем ошибку
      break;
    case WSAESOCKTNOSUPPORT:  // здесь обрабатываем ошибку
      break;
    default:        // такая ситуация не должна возникать вообще
      break;
  }
}
```

```
// Listing 1.02 nix

sd = socket (PF_INET, SOCK_DGRAM, 0);

if (sd == -1)
{
  switch (errno)
  {
    case EPROTONOSUPPORT:    // здесь обрабатываем ошибку
      break;
    case EMFILE:      // здесь обрабатываем ошибку
      break;
    case ENFILE:      // здесь обрабатываем ошибку
      break;
    case EACCES:      // здесь обрабатываем ошибку
      break;
    case ENOBUFS:      // здесь обрабатываем ошибку
      break;
    case EINVAL:      // здесь обрабатываем ошибку (*только для Linux)
      break;
    default:        // такая ситуация не должна возникать вообще
      break;
  }
}
```

Теперь мы можем составить подобие таблицы ошибок, их описаний и рекомендаций к действиям:

Linux & FreeBSD:

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

- возвращает целое неотрицательное число, в случае успешного завершения, или -1 при ошибке и устанавливает errno равным коду ошибки. errno может быть равна:

EPROTONOSUPPORT

- данный протокол не поддерживается в текущем домене, есть как минимум четыре варианта развития событий:
исправляем опечатку или неточность в коде - и все работает
бросаем это дело и ждем, когда поддержка протокола будет реализована
сами создаем код, осуществляющий поддержку этого протокола
используем другой протокол

EMFILE

- таблица процессов перегружена, опять же есть несколько вариантов действий:
избавление станции от лишних поглотителей ресурсов системы и повтор
аварийный выход
ожидание освобождения таблицы и повтор

ENFILE

- недостаточно памяти для выполнения операции, боремся так:
тщательно проверяем свой код на наличие "утечки памяти"
ждем освобождения ресурсов и повторяем попытку
освобождаем ресурсы насильно и повторяем попытку :)
выходим аварийно

EACCESS

- нет прав для выполнения операции, боремся так:
крадем, нелегально получаем эти права
легально получаем эти права
обижаемся и тихо забиваемся в угол
используем средства, не требующие наличия данных прав

EINVAL

- неподдерживаемый (или неизвестный) протокол (*присутствует только в Linux)

ENOBUFS

- недостаточно памяти для размещения буферов, сокет не может быть создан в текущих условиях, бороться с этой бедой так же, как и с ENFILE

Windows:

```
// winsock2.h: typedef u_int SOCKET


SOCKET socket (int pf, int type, int protocol);
```

- возвращает целое положительное число, в случае успешного завершения, или 0 в случае ошибки. Результат ошибки может быть получен при помощи вызова int WSAGetLastError (void); Ошибки могут быть следующего вида:

WSANOTINITIALISED

- перед вызовом socket() надо успешно завершить вызов WSAStartup(), бороться с этим можно только одним способом:
лечим склероз

WSAENETDOWN

- ошибка в сетевой подсистеме:
проверяем, правильно ли настроено оборудование и функционирует ли оно

WSAEAFNOSUPPORT

- указанный домен не поддерживается (куда деваться? сидим и ждем)

WSAEINPROGRESS

- выполняется блокирующий вызов winsock 1.1 (мы коснемся этой темы позже)

WSAEMFILE

- нет свободных дескрипторов сокетов, боремся так же как и в *nix

WSAENOBUFS

- нет памяти для размещения буферов, те же рекомендации

WSAEPROTONOSUPPORT

- указанный протокол не поддерживается, ждем известий о реализации поддержки...

WSAEPROTOTYPE

- указан протокол, не соответствующий типу сокета:
перечитываем это руководство 20 раз

WSAESOCKTNOSUPPORT

- данный тип сокета не поддерживается в указанном домене, ждем...

Все предельно ясно, за исключением некоторых тонких моментов, которые мы подробно разберем чуть позже. В любом случае, советую обратиться либо к MSDN, либо к man pages.

Итак, создавать сокеты мы уже умеем. Но эти сволочи в процессе своей жизнедеятельности мусорят за троих, поэтому нам надо научиться за ними убирать. Реализация winsock требует, чтобы каждому вызову WSAStartup соответствовал вызов WSACleanup(). Однако, для чистоты в доме нам не только надо убрать мусор, который оставила после себя библиотека, но и также убраться за собой (сокет создавали мы, значит, и убивать должны мы же). Вот как это выглядит:

Windows:

```
int closesocket (SOCKET sd);
int WSACleanup (void);
```

Linux & FreeBSD:

```
int close (int sd);
```

А теперь рассмотрим весь процесс очистки под обеими системами (включая проверку ошибок):

```
// Listing 1.03 win

int wsError = closesocket (sd);
if (wsError == SOCKET_ERROR)    // winsock2.h: #define SOCKET_ERROR (-1)
```

```
{
  wsError = WSAGetLastError();
  switch (wsError)
  {
    case WSANOTINITIALISED:   // здесь обрабатываем ошибку
      break;
    case WSAENETDOWN:    // здесь обрабатываем ошибку
      break;
    case WSAEINTR:      // здесь обрабатываем ошибку
      break;
    case WSAEINPROGRESS:    // здесь обрабатываем ошибку
      break;
    case WSAENOTSOCK:    // здесь обрабатываем ошибку
      break;
    case WSAEWOULDBLOCK:    // здесь обрабатываем ошибку
      break;
    default:      // такая ситуация не должна возникать вообще
      break;
  }
}
wsError = WSACleanup ();
if (wsError == SOCKET_ERROR)
{
  wsError = WSAGetLastError();
  switch (wsError)
  {
    case WSANOTINITIALISED:   // здесь обрабатываем ошибку
      break;
    case WSAENETDOWN:    // здесь обрабатываем ошибку
      break;
    case WSAEINPROGRESS:    // здесь обрабатываем ошибку
      break;
    default:      // такая ситуация не должна возникать вообще
      break;
  }
}
```

```
}

// Listing 1.03nix

int result = close (sd);
if (result == -1)
{
  switch (errno)
  {
    case EBADF:       // здесь обрабатываем ошибку
      break;
    case EINTR:      // здесь обрабатываем ошибку
      break;
    case EIO:      // здесь обрабатываем ошибку (*только Linux)
      break;
    default:      // такая ситуация не должна возникать вообще
      break;
  }
}
```

Первое, что бросается в глаза в коде Windows - это появление новых констант, определяющих новые виды ошибок: WSAENOTSOCK, WSAEINTR и WSAEWOULDBLOCK. На данный момент нас интересует только первая WSAENOTSOCK (означает, что в функцию closesocket() был передан не дескриптор сокета, а значит, нам надо быть внимательней с похмелья). Объяснение оставшихся двух констант пока отложим на неопределенный срок.

Что касается кода *nix - то EBADF - это аналог WSAENOTSOCK, EINTR - означает, что во время выполнения вызова был получен сигнал, который прервал выполнение вызова. EIO - свидетельствует об ошибке ввода-вывода.

Вообще говоря, ты, уважаемый читатель уже заметил, что мы раз от разу проверяем выполнение на возникновение одних и тех же ошибок, поэтому было бы проще сразу объединить все константы в одной пользовательской функции и вызывать ее для проверки ошибок (просто для компактности кода, сокращения switch-конструкций и т.д.).

В данном каркасе мы не осуществляем непосредственной пересылки данных. Мы просто все готовим, смотрим, что получилось, потом убираемся за собой и уходим. Реализация winsock требует, чтобы мы очищали всю использованную память ее методами. Для этого мы вызываем две функции, первая (closesocket()) убивает наш дескриптор (почти в прямом смысле слова, то есть closesocket() немедленно завершает все операции, стоящие в

очереди на выполнение сокетом), вторая же производит очистку всей памяти, которая была выделена для нормального функционирования winsock (эта реализация скрыта от наших, глаз). Если же мы не в Windows-системе, то мы просто закрываем сокет классическим способом, используя системный вызов close(). Надо заметить: каждому вызову socket() должен быть поставлен в соответствие либо вызов closesocket() (Windows), либо close() (*nix). А также каждому вызову WSAStartup() должен соответствовать вызов WSACleanup(). Все очень просто: одна функция делает мусор, вторая убирает. Мы намеренно используем жесткий и немедленный вызов closesocket() в Windows-части кода, потому что нас на данный момент не интересует, были ли отосланы все данные находящиеся в очереди, были ли приняты данные, ожидающие во входящей очереди, были ли выполнены другие операции, присущие некоторым сокетам и т.п. Забегая вперед, скажу, что есть методы, позволяющие контролировать процесс закрытия сокета (то есть делать все мягко и элегантно - сказать "пока, милая!" и повесить трубку, а не обрубать телефонный провод топором), но нам до него еще пока очень далеко :).

Люди, как правило, выбирают одну из платформ для программирования своих приложений и ориентируются на нее. Кому-то нравится клиентская непринужденность Windows-систем, кому-то хочется серверной мощи *nix (а может ты вообще сидишь сейчас над Solaris/SPARC?? :)) Как бы там ни было, лично я предпочитаю FreeBSD/x86.

Страницы: 1 **2** 3 Следующая »

#OSI, #сокеты

18 мая 2003 (Обновление: 20 янв 2011)

Комментарии [4]

Контакт
Сообщества
Участники
Каталог сайтов
Категории
Архив новостей