Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра инженерной психологии и эргономики

ИНТЕРФЕЙС В СИСТЕМАХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Лабораторный практикум
для студентов специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий»
всех форм обучения

Минск БГУИР 2011

УДК 004.51:004(076.5)
ББК 32.973стд1-018.2я73
И73

С о с т а в и т е л и:

И. В. Байдаков, В. С. Осипович, К. Д. Яшин

Р е ц е н з е н т:

доцент кафедры информатики учреждения образования
«Белорусский государственный университет информатики
и радиоэлектроники», кандидат физико-математических наук С. И. Сиротко

Интерфейс в системах информационных технологий : лаб. практикум
И73 для студ. спец. 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий» всех форм обуч. / сост. И. В. Байдаков, В. С. Осипович, К. Д. Яшин. – Минск : БГУИР, 2011. – 38 с. : ил.
ISBN 978-985-488-533-7.

Содержит краткие теоретические сведения, примеры простейших программ и методические указания по выполнению лабораторных работ по дисциплине «Интерфейс в системах информационных технологий».

Предназначен для студентов и преподавателей, владеющих теоретическим материалом по темам предлагаемых лабораторных работ.

Версию практикума, дополненную теоретическим материалом, можно получить в электронном варианте на кафедре инженерной психологии и эргономики.

УДК 004.51:004(076.5)
ББК 32.973стд1-018.2я73

ISBN 978-985-488-533-7

СОДЕРЖАНИЕ

Лабораторная работа №1
ЗНАКОМСТВО С СЕТЕВЫМИ СОКЕТАМИ

Цель работы: формирование практических навыков в работе с базовыми сетевыми функциями сокетов.

1.1. Теоретические сведения. Для подключения к серверу клиент должен знать его адрес и предоставлять ему свой. Чтобы обмениваться сообщениями независимо от своего местоположения, клиент и сервер используют сокеты. Обратимся к примеру с телефонным звонком. Телефонная трубка имеет два основных элемента: микрофон (передатчик) и динамик (приемник). Телефонный номер по сути представляет собой уникальный адрес. У сокета имеются также два канала: один для прослушивания, а другой для передачи (подобно каналам ввода-вывода в файловой системе). Клиент (звонящий) подключается к серверу (абоненту), чтобы начать «сетевой разговор». Каждый участник «разговора» предлагает несколько стандартных, заранее известных сервисов, например, телефон, по которому можно узнать правильное время.

Клиентская программа должна предпринять ряд действий для установления соединения с другим компьютером или сервером. Причем эти действия следует выполнять в определенной последовательности. Последовательность действий имеет следующий вид: создание сокета, поиск адресата, организация канала связи с другой программой и разрыв соединения (рис. 1.1).

Программирование сокетов отличается от прикладного и инструментального программирования, поскольку приходится иметь дело с одновременно выполняющимися программами. Значит, требуется дополнительно решать вопросы синхронизации и управления ресурсами. Сокеты позволяют асинхронно передавать данные через двунаправленный канал. При этом могут возникать различного рода проблемы, например взаимоблокировки процессов и зависания программ. При тщательном проектировании приложений большинство таких проблем вполне можно избежать.

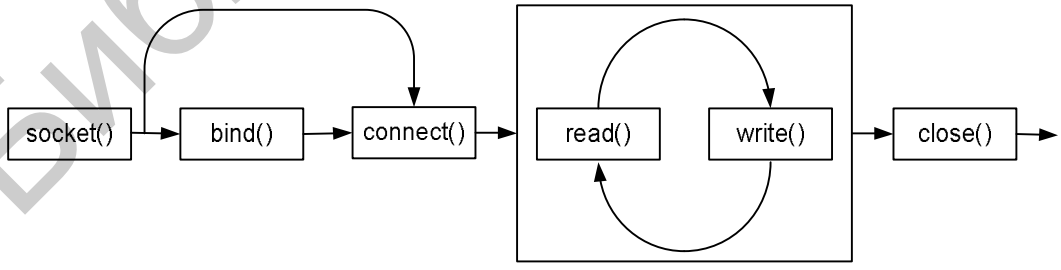

Рис. 1.1. Порядок вызова сетевых функций операционной системы клиентом

Простейший алгоритм работы клиентской программы. Простейшим соединением является такое соединение, в котором клиент подключается к серверу, посылает запрос и получает ответ. Некоторые стандартные сервисы

даже не требуют наличия запроса, например, сервис текущего времени, доступный через порт с номером 13. Во многих системах Linux этот сервис по умолчанию недоступен и, чтобы иметь возможность обращаться к нему, требуется модифицировать файл /etc/inetd.conf. Если у вас есть доступ к компьютеру, работающему под управлением операционной системы BSD, HP-UX или Solaris, попробуйте обратиться к указанному порту. Есть несколько сервисов, к которым можно получить доступ. Запустите программу Telnet и свяжитесь с портом 21 (FTP): % telnet 127.0.0.1 21.

Описанный алгоритм может показаться чересчур упрощенным. В принципе так оно и есть. Но сама процедура подключения к серверу и организации взаимодействия с ним действительно проста. Ниже рассматривается каждый из указанных выше этапов.

Системный вызов socket(). Синтаксис функции таков:

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

В примерах, приведенных в данной лабораторной работе, будут использоваться следующие параметры (табл. 1.1): domain=PF_INET, type=SOCK_STREAM, protocol=0.

Вызов протокола TCP выглядит следующим образом:

```
int sd;
sd = socket(PF_INET, SOCK_STREAM,  0);
```

В переменную sd будет записан дескриптор сокета, функционально эквивалентный дескриптору файла:

```
int fd;
fd = open(...);
```

Таблица 1.1

Избранные параметры фунции socket()

Параметр	Значение	Описание
1	2	3
domain	PF_INET	Протоколы семейства IPv4; стек TCP/IP
	PF_LOCAL	Локальные именованные каналы в стиле BSD; используется утилитой журнальной регистрации, а также при организации очередей принтера
	PF_IPX	Протоколы Novell
	PF_INET6	Протоколы семейства IPv6; стек TCP/IP
type	SOCK_STREAM	Протокол последовательной передачи данных (в виде байтового потока) с подтверждением доставки (TCP)
	SOCK_RDM	Протокол пакетной передачи данных с подтверждением доставки (еще не реализован в большинстве операционных систем)
	SOCK_DGRAM	Протокол пакетной передачи данных без подтверждения доставки (UDP – User Datagram Protocol)

Окончание табл. 1.1

1	2	3
	SOCK_RAW	Протокол пакетной передачи низкоуровневых данных без подтверждения доставки
protocol		Представляет собой 32-разрядное целое число с сетевым порядком следования байтов. В большинстве типов соединений допускается единственное значение данного параметра: 0 (нуль), а в соединениях типа sock_raw параметр должен принимать значения от 0 до 255

Следует знать о том, какие файлы заголовков требуется включать в программу. В Linux они таковы:

```
    #include <sys/socket.h>   /* содержит прототипы функций */
    #include <sys/types.h> /*содержит объявления стандартных системных типов
данных*/
    #include <resolv.h>       /* содержит объявления дополнительных типов данных
*/
```

В файле sys/socket.h находятся объявления функций библиотеки Socket A (включая функцию socket(), естественно). В файле sys/types.h определены многие типы данных, используемых при работе с сокетами.

Подключение к серверу. Синтаксис функции connect() таков:

```
#include <sys/socket.h>
#include <resolv.h>
int connect(int sd,  struct sockaddr *server, int addr_len);
```

Первый параметр (sd) представляет собой дескриптор сокета, который был создан функцией socket(). Последний, третий, параметр задает длину структуры sockaddr, передаваемой во втором параметре, так как она может иметь разные тип и размер. Такой синтаксис делает функции socket() принципиально отличной от синтаксиса функций файлового ввода-вывода.

Приведем общий вид структуры адреса и рядом для сравнения структуру адреса в домене PF_INET (взято из файлов заголовков):

```
struct sockaddr {                    struct sockaddr_in {
unsigned short int  sa_family;         sa_family_t       sin_family;
unsigned char       sa_data[14];       unsigned short int      sin_port;
}                                      struct in_addr    sin_addr;
                                       unsigned char     __pad[];  };
```

Описание полей структуры sockaddr_in см. в табл. 1.2.

Прежде чем вызвать функцию connect(), программа должна заполнить описанные поля. В нижеследующем листинге показано, как это сделать. Вообще говоря, в Linux не требуется приводить структуру sockaddr_in к типу sockaddr. Если же предполагается использовать программу в разных системах, можно легко добавить операцию приведения типа.

```
#define PORT_TIME 13
struct sockaddr_in dest;
char *host = "127.0.0.1";
int sd;             /**** создание сокета ****/
bzero(&dest,  sizeof(dest));                /* обнуляем структуру */
dest.sin_family = AF_INET;                  /* выбираем протокол */
dest.sin_port = htons(PORT_TIME);           /*  выбираем порт */
inet_aton(host, &dest.sin_addr);                 /*  задаем адрес */
if (connect(sd,sdest,sizeof(dest)) != 0)         /* подключаемся */
```

```
{   perror("socket connection");
  abort();          }   ...
```

Таблица 1.2

Описание полей структуры sockaddr_in

Поле	Описание	Порядок байтов	Пример
sin_family	Семейство протоколов	Серверный	AF_INET
sin_port	Номер порта сервера	Сетевой	13
sin_addr	IP-адрес сервера	Сетевой	127.0.0.1

Получение ответа от сервера. Когда сокет открыт, можно вызывать стандартные низкоуровневые функции ввода-вывода для приема/передачи. Объявление функции read():

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Эта функция должна быть вам знакома. Вы много раз применяли ее при работе с файлами, только на этот раз необходимо указывать дескриптор не файла (fd), а сокета (sd). Вот, как обычно организуется вызов функции read():

```
int sd, bytes_read;
sd = socket(PF_INET, S0CK_STREAM, 0);           /* создание сокета */
/***... подключение к серверу ***/
bytes_read = read(sd, buffer, MAXBUF);          /* чтение данных */
if ( bytes_read < 0 ) ...           /*  сообщить  об  ошибках;  завершить
работу */
```

Дескриптор сокета можно даже преобразовать в файловый дескриптор (FILE*), если требуется работать с высокоуровневыми функциями ввода-вывода. Например, в следующем фрагменте программы демонстрируется, как применить функцию fscanf() для чтения данных с сервера:

```
char Name[NAME], Address[ADDRESS], Phone[PHONE];
FILE *sp;               /*высокоуровневая   структура   -   обертка   над
сокетом*/
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);      /**** подключение к серверу ****/
if (  (sp = fopen(sd,  "r")) == NULL )
  perror("FILE* conversion failed");
else if (  fscanf(sp,  "%*s,  %*s, %*s\n", NAME, Name,
  ADDRESS, Address, PHONE,  Phone)  < 0)
{ perror("fscanf"); ...
```

Функция read() не содержит информации о том, как работает сокет. В Socket API есть другая функция – recv(), которая наряду с чтением данных позволяет контролировать работу сокета:

```
#include <sys/socket.h>
#include <resolv.h>
int recv(int sd, void *buf, int len, unsigned int flags);
```

Функция recv() является более гибкой, чем read(). Прочесть данные из канала сокета (эквивалентно функции read()) можно следующим образом:

```
int bytes_read;
bytesjread = recv(sd, buffer, MAXBUF,  0);
```

Неразрушающее чтение осуществляется так:

```
int bytes_read;
bytesjread = recv(sd, buffer, MAXBUF, MSG_PEEK);
```

Задание режима неразрушающего чтения внеполосных данных:

```
int bytes_read;
bytes_ead = recv(sd, buffer, MAXBUF, MSG_OOB |  MSG_PEEK);
```

В первом фрагменте функция просто передает серверу указатель буфера и значение его длины. Во втором фрагменте информация копируется из очереди, но не извлекается из нее. В третьем фрагменте внеполосные данные копируются из очереди, но не извлекаются из нее. Во всех трех фрагментах есть одно преднамеренное упущение. Что если сервер пошлет больше информации, чем может вместить буфер? В действительности ничего страшного не произойдет, это не критическая ошибка. Просто программа потеряет те данные, которые не были прочитаны.

Разрыв соединения. Информация от сервера получена, сеанс прошел нормально. Следует прекратить связь. Есть два способа сделать это. В большинстве программ используется стандартный системный вызов close():

```
#include <unistd.h>
int close(int fd);
```

С помощью функции shutdown() можно закрыть канал в одном направлении, сделав его доступным только для чтения или только для записи:

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

Параметр how может принимать значения, представленные в табл. 1.3.

Таблица 1.3

Значение аргумента how для функции shutdown()

Значение	Выполняемое действие
0	Закрывает канал чтения
1	Закрывает канал записи
2	Закрывает оба канала

Порядок следования байтов. Предположим, имеется число 214259635. В шест-надцатеричном виде оно записывается как OxOCC557B3. Процессор с обратным порядком байтов хранит его следующим образом:

```
Адрес:    00   01   02   03   04
Данные:   OC   C5   57   B3   ...
```

Обратите внимание на то, что старший байт (0C) указан первым. Процессор с прямым порядком байтов хранит число по-другому:

```
Адрес:   00   01   02   03   04
Данные:  B3   57   C5   OC   ...
```

Теперь первым указан младший байт (B3).

Функции преобразования данных. Существует ряд средств, позволяющих выполнить соответствующее преобразование. Они находят широкое применение в сетевых приложениях, заполняющих поля структур семейства sockaddr. Некоторые из них представлены в серверном порядке (см. табл. 1.2 и 1.4).

Таблица 1.4

Функции преобразования порядка следования байтов

Функция	Преобразование	Описание
htons()	Из серверного порядка в сетевой короткий	Представляет 16-разрядное число в обратном порядке
htonl()	Из серверного порядка в сетевой длинный	Представляет 32-разрядное число в обратном порядке
ntohs()	Из сетевого порядка в серверный короткий	Представляет 16-разрядное число в серверном порядке
ntohl()	Из сетевого порядка в серверный длинный	Представляет 32-разрядное число в серверном порядке

Функция inet_aton() преобразует IP-адрес из точечной записи формата ASCII в эквивалентную двоичную форму, представленную в сетевом порядке (т. е. после нее не требуется вызывать функцию htonl()). Эта функция также входит в группу функций преобразования (табл. 1.5). Теперь перейдем к написанию серверной части.

Схема работы сокета. Схематически общий алгоритм сервера представлен на рис. 1.2. Рассмотрим пример реализации простого эхо-сервера. Это основа основ серверного программирования, подобно приложению «Hello, World» в программировании на языке С. Большинство соединений можно проверить, послав данные и запросив их назад в неизменном виде (эхо). Аналогичным образом пишутся и отлаживаются даже самые сложные приложения.

Таблица 1.5

Функции преобразования адреса

Функция	Описание
inet_aton()	Преобразует адрес из точечной записи (###.###.###.###) в двоичную форму с сетевым порядком следования байтов; возвращает нуль в случае неудачи и ненулевое значение, если адрес допустимый
inet_addr()	Устарела (аналог inet aton()), так как неправильно обрабатывает ошибки; при возникновении ошибки возвращает единицу (хотя 255.255.255.255 – обычный широковещательный адрес)
inet_ntoa()	Преобразует IP-адрес, представленный в двоичном виде с сетевым порядком следования байтов, в точечную запись формата ASCII
gethostbyname()	Просит сервер имен преобразовать имя (такое, как www.linux.org) в один или несколько IP-адресов

В общем случае в серверной программе требуется в определенной последовательности вызвать ряд системных функций. На примере эхо-сервера можно наглядно увидеть эту последовательность, не отвлекаясь на решение других, более специфических задач. Общий алгоритм работы эхо-сервера: 1. Создание сокета с помощью функции socket(). 2. Привязка к порту с помощью

функции bind(). 3. Перевод сокета в режим прослушивания с помощью функции listen(). 4. Проверка подключения с помощью функции accept(). 5. Чтение сообщения с помощью функции recv() или read(). 6. Возврат сообщения клиенту с помощью функции send() или write(). 7. Если полученное сообщение не является строкой «bye», возврат к п. 5. 8. Разрыв соединения с помощью функции close() или shutdown(). 9. Возврат к п. 4.

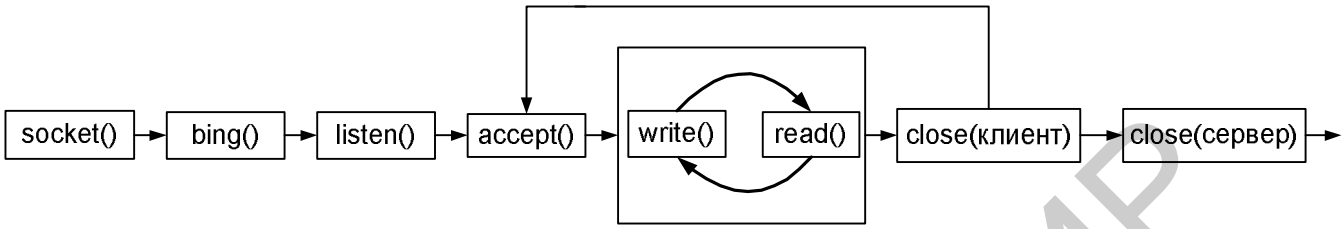

Рис. 1.2. Порядок вызова сетевых функций операционной системы сервером

В приведенном алгоритме четко видны особенности работы с протоколом TCP в отличие от работы с протоколом UDP и другими протоколами, не ориентированными на установление соединений. Здесь сервер не закрывает соединение до тех пор, пока клиент не пришлет команду bye.

Привязка порта к сокету. Работа с TCP-сокетами начинается с вызова функции socket(), которой передается константа SOCK_STREAM. Теперь требуется задать также номер порта, чтобы клиент мог к нему подключиться.

Функция bind() объявляется так:

```
#include <sys/socket.h>
#include <resolv.h>
int bind(int sd, struct sockaddr *addr, int addr_size);
```

Перед вызовом функции bind() необходимо заполнить поля структуры sockaddr:

```
struct sockaddr_in addr;                    /* создаем TCP-сокет    */
bzero(&addr, sizeof(addr));                 /* обнуляем структуру   */
addr.sin_family = AF_INET;                  /* выбираем стек TCP/IP*/
addr.sin_port = htons(MY_PORT);             /* задаем номер порта   */
addr.sin_addr.s_addr = INADDR_ANY;          /* любой IP-адрес */
if ( bind(sd,&addr,sizeof(addr))!=0)        /* запрашиваем порт     */
  perror("Bind AF_INET");
```

Создание очереди ожидания. Очередь сокета активизируется при вызове функции listen(). Когда сервер вызывает эту функцию, он указывает число позиций в очереди. Кроме того, сокет переводится в режим «Только прослушивание». Это очень важно, так как позволяет впоследствии вызывать функцию accept():

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

Функция listen() может генерировать следующие ошибки: 1. EBADF – указан неверный дескриптор сокета. 2. EOPNOTSUPP – протокол сокета не поддерживает функцию listen(). В TCP (SOCK_STREAM) очередь ожидания поддерживается, а в протоколе UDP (SOCK_DGRAM) – нет.

Прием запросов от клиентов. Можно узнать, кто устанавливает соединение с сервером, поскольку в функцию accept() передается информация о клиенте:

```
#include <sys/socket.h>
#include <resolv.h>
int accept(int sd, sockaddr *addr, int *addr_size);
```

Приведем пример использования функции accept():

```
int sd;
struct sockaddr_in addr;
/*** ... создание сокета, привязка его к порту и перевод в режим прослушивания
***/
for (;;)                          /* цикл повторяется бесконечно */
{
  int clientsd;                        /* новый дескриптор сокета */
  int size = sizeof(addr);                  /* вычисление размера структуры
*/
  clientsd = accept(sd, &addr, &size);      /* ожидание подключения */
  if ( clientsd > 0 )                       /* ошибок нет */
             /*** Взаимодействие с клиентом ***/
             { close(clientsd); }           /* очистка и отключение */
  else                            /* произошла ошибка */
  perror("Accept");    }
```

Взаимодействие с клиентом. Большинство полей структуры имеет сетевой порядок следования байтов. Извлечь адрес и номер порта из переменной addr можно с помощью функций преобразования. Приведём пример использования функции accept(), информация о каждом новом подключении отображается на экране:

```
/* … внутри цикла */
client = accept(sd, &addr, &size);
if ( client > 0 )
{ if ( addr.sin_family == AF_INET)
        printf("Connection[%s]: %s:%d\n",         /* регистрация */
        ctime(time(0)),                           /* метка времени */
        ntoa(addr.sin_addr),
        ntohs(addr.sin_port)); }                  /* взаимодействие с клиентом */
```

Приведем окончание реализации эхо-сервера:

```
/* внутри цикла после вызова функции accept() */
if ( client > 0 ) {
  char buffer[1024]; int nbytes;
  do
  {     nbytes = recv(client, buffer, sizeof(buffer), 0);
        if ( nbytes > 0 )             /* если получены данные, возвращаем их
*/
             send(client, buffer, nbytes, 0);    }
  while ( nbytes > 0 && strncmp("bye\r", buffer, 4) != 0);
  close(client);    }
```

Особенности использования Socket API под операционными системами семейства Windows. Для использования Socket API следует подключать заголовочный файл winsock2.h и использовать библиотеку ws2_32.lib. Перед использованием сетевых функций под Windows следует инициализировать библиотеку работы с сокетами. Это делается так:

```
WSADATA WsaDat;
WSAStartup(MAKEWORD(2,2),&WsaDat);,
```

где первый аргумент задает номер используемой версии библиотеки Winsock. В данном случае используется версия 2.2. В случае успешной инициализации функция возвращает нулевое значение.

В конце исполнения программы следует оповестить библиотеку Winsock о том, что она больше не будет использоваться, для того чтобы она смогла освободить системные ресурсы. Делается это с помощью функции WSACleanup():

```
WSACleanup();
```

В остальном работа с сокетами под операционной системой Windows не отличается от работы с другими реализациями Socket API.

1.2. Порядок выполнения работы. Реализовать простейший «клиент» и «сервер» по передаче файлов по сети. Использовать потоковые сокеты. Свой протокол по передачи файла придумывать не нужно: считать, что по входящему соединению сразу же передается файл, а закрытие соединения интерпретировать как конец файла. Для удобства рекомендуется передавать адрес и номера портов в качестве аргументов командной строки.

1.3. Контрольные вопросы. 1. Что такое сокет? **2.** Приведите простейшую схему работы клиента. **3.** Приведите простейшую схему работы сервера. **4.** В чём отличие структуры sockaddr от структуры sockaddr_in? Для чего это сделано? **5.** Что делает функция accept()?

Лабораторная работа №2
ОСВОЕНИЕ ОРИЕНТИРОВАННЫХ НА СОЕДИНЕНИЕ СОКЕТОВ (TCP)

Цель работы: формирование практических навыков в работе с ориентированными на соединение сокетами (TCP).

2.1. Теоретические сведения. ***Клиент-серверная архитектура.*** Обычно клиенты и серверы осуществляют взаимодействие через компьютерную сеть, т.е. на разных аппаратных платформах. Сервер часто представляет собой высокопроизводительную машину, на которой выполняются одна или несколько серверных программ, которые «делятся» своими ресурсами с клиентами; клиенты же запрашивают эти ресурсы, вызывая определенные серверные функции. Таким образом, клиенты инициируют коммуникацию с сервером (активная роль), в то время как серверы ожидают соединений для их обслуживания (пассивная роль).

Клиент-серверная модель является одной из центральных в распределенной обработке информации. На ее основе сконструированы такие протоколы Интернета, как HTTP, SMTP, telnet, DNS.

Особенности работы сервера. В случае если сервер имеет много клиентов, то они будут простаивать в период обслуживания первого клиента, т.е. появляются дополнительные задержки в обслуживании, свидетельствующие о низком качестве обслуживания сервером.

Концептуальное решение этой проблемы следующее: до актуального чтения данных из сокета, потенциально вызывающего длительную блокировку сервера, нужно опросить сокет на предмет того, имеются ли у него данные и могут ли они быть прочитаны без блокирования. Программное решение составляют функции poll() и select().

Функция poll() имеет следующий прототип:

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Интерпретировать возвращаемое значение следует так. Если возвращаемое значение меньше нуля, значит, произошла ошибка. Нулевое значение свидетельствует о том, что истек период ожидания. В случае успешного завершения возвращается число каналов, в которых произошли изменения.

Параметр usdf указывает на массив структур pollfd, которые в свою очередь задаются так:

```
struct pollfd  {
  int fd;                       /* дескриптор файла или сокета */
  short events;                 /* запрашиваемые события */
  short revents; };             /* события, которые произошли в канале */
```

В поле fd содержится проверяемый дескриптор файла. Поля events и revents обозначают соответственно проверяемые и произошедшие события. События задаются следующими константами: 1. POLLIN – поступили данные. 2. POLLPRI – поступили срочные (внеполосные) данные. 3. POLLOUT – канал готов для записи. 4. POLLERR – произошла ошибка. 5. POLLHUP – отбой на другом конце канала. 6. POLLNVAL – неправильный запрос, канал fd не был открыт. 7 POLLRDNORM – обычное чтение (только в Linux). 8. POLLRDBAND – чтение внеполосных данных (только в Linux). 9. POLLWRNORM – обычная запись (только в Linux).

Возможны следующие ошибки: 1. ENOMEM – недостаточно памяти для создания записей в таблице дескрипторов файлов. 2. EFAULT – указатель ссылается на область памяти, выходящую за пределы адресного пространства процесса. 3. EINTR – получен сигнал до того, как произошло одно из запрашиваемых событий.

Приведем пример использования функции poll():

```
 int fd_count=0;
struct pollfd fds[MAXFDs];
fds[fd_count].fd = socket(PF_INET, SOCK_STREAM, 0);
/*** вызовы функций bind() и listen() ***/
fds[fd_count++].events = POLLIN;
for (;;) {
  if ( poll(fds, fd_count, TIMEOUT_MS) > 0 ) {
        int i;
        if ( (fds[0].revents & POLLIN) != 0 ) {
              fds[fd_count].events = POLLIN | POLLHUP;
              fds[fd_count++].fd = accept(fds[0].fd, 0, 0); }
        for ( i = 1; i < fd_count; i++ ) {
              if ( (fds[iJ.revents & POLLHUP) != 0 ) {
                    close(fds[i].fd);
              /*** перемещаем дескрипторы для заполнения пустых позиций ***/
                    fd_count--;                    }
              else if ( (fds[i].revents & POLLIN) != 0 )           }      }
}
              /*** читаем и обрабатываем данные ***/
```

Функция select() ожидает изменения статуса одного из заданных каналов ввода-вывода. Когда в одном из каналов происходит изменение, функция завершается. Имеется четыре макроса, предназначенных для управления набором дескрипторов: 1. FD_CLR – удаляет дескриптор из списка. 2. FD_SET – добавляет дескриптор в список. 3. FD_ISSET – проверяет готовность канала к выполнению операции ввода-вывода. 4. FD_ZER0 – инициализирует список дескрипторов.

Функция select() и связанные с ней макросы имеют следующий прототип:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int hi_fd, fd_set *readfds, fd_set *writefds,
  fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

При успешном завершении функция возвращает число каналов, в которых произошли изменения. В случае ошибки возвращается отрицательное значение. Если превышен допустимый период ожидания, возвращается нуль.

Возможны следующие ошибки: 1) EBADF – в один из списков входит неправильный дескриптор; 2) EINTR – получен неблокируемый сигнал; 3) EINVAL – указан отрицательный размер списка; 4) ENOMEM – не хватает памяти для создания внутренних таблиц. Приведём пример использования функции select():

```
int i, ports[]={9001, 9002, 9004, -1};
int sockfd, max=0;
fd_set set;
struct sockaddr_in addr;
struct timeval timeout={2,500000};                /* 2,5 секунды */
FD_ZERO(&set);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
for ( i = 0; ports[i] > 0; i++) {
  sockfd = socket(PF_INET, SOCK_STREAM, 0);
  addr.sin_port = htons(ports[i]);
  if ( bind(sockfd, &addr, sizeof(addr)) != 0 )
        perror("bind() failed");
  else {
        FD_SET(sockfd, &set);
        if ( max < sockfd ) max = sockfd;   }     }
if ( select(max+l, &set, 0, &set, &timeout) > 0 ) {
  for ( i = 0; i <= max; i++)
        if ( FD_ISSET(i, &set) ){
              int client = accept(i, 0, 0); } }   /***обрабатываем клиентский
запрос***/
```

2.2. Порядок выполнения работы. Воспользовавшись клиент-серверной архитектурой, реализовать чат в локальной сети по протоколу TCP: имеется N клиентов (клиентских программ), которым при старте задаются следующие параметры: псевдоним, хост с чат-сервером (IP-адрес) и номер обслуживающего порта. Когда один клиент что-либо пишет в чате, все другие должны увидеть то, что он напечатал. Нужно разработать простейший протокол

для коммуникации клиента и сервера (где, например, будут содержаться псевдоним, время отправки сообщения и сам текст сообщения).

2.3. Контрольные вопросы. 1. Что представляет собой клиент-серверная архитектура? **2.** Перечислите преимущества и недостатки клиент-серверной архитектуры. **3.** Для чего нужны функции poll() и select()?

Лабораторная работа №3
ОСВОЕНИЕ СОКЕТОВ, НЕ ОРИЕНТИРОВАННЫХ НА СОЕДИНЕНИЕ (UDP)

Цель работы: формирование практических навыков в работе с не ориентированными на соединение сокетами (UDP), освоение концепции широковещания в сети.

3.1. Теоретические сведения. ***Широковещательный механизм.*** Когда один из компьютеров (или маршрутизаторов) включается, то он должен оповестить локальную сеть о своем появлении. Но как он может это сделать, не зная IP-адресов своих соседей? Поэтому он рассылает широковещательный пакет, который говорит «Привет всем!». В ответ на этот пакет, например, каждый компьютер в локальной сети также отвечает лично ему «Привет!». Из этого ответа наш включенный компьютер может извлечь информацию о других компьютерах (во-первых, он узнает сам факт их существования, а во-вторых, их IP-адреса).

Основы протокола UDP. Среди набора протоколов Интернета есть транспортный протокол без установления соединения – UDP (User Datagram Protocol – пользовательский дейтаграммный протокол). UDP позволяет приложениям отправлять инкапсулированные IP-дейтаграммы без установления соединений. UDP описан в RFC 768. С помощью протокола UDP передаются сегменты, состоящие из 8-байтного заголовка, за которым следует поле полезной нагрузки. Заголовок показан на рис. 3.1.

Рис. 3.1. Заголовок UDP-пакета

Рассмотрим возможности Socket API по работе с сокетами, не ориентированными на соединение. Если для передачи данных соединение не устанавливается, то и функцию connect() вызывать не нужно. Однако в этом случае нельзя будет вызвать функции send() и recv(). В операционной системе существуют две аналогичные функции, позволяющие задавать адрес получателя, – sendto() и recvfrom(), – которые служат для отправки и приема соответственно. Они имеют следующие прототипы:

```
#include <sys/socket.h>
#include <resolv.h>
int sendto(int sd, char* buffer, int msg_len,
  int options, struct sockaddr *addr, int addr_len);
int recvfrom(int sd, char* buffer, int maxsize, int options,
  struct sockaddr *addr, int *addr_len);
```

Первые четыре параметра такие же, как и параметры в функциях send() и recv(). Даже опции и возможные коды ошибок совпадают. В функции sendto() дополнительно указывается адрес получателя. При отправке сообщения необходимо заполнить поля структуры sockaddr. Приведем пример:

```
int sd;
struct sockaddr_in addr;
sd = socket(PF_INET, SOCK_DGRAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST PORT);
inet_aton(DEST_ADDR, &addr.sin_addr);
sendto(sd, "This is a test", 15, 0, &addr, sizeof(addr));
```

В примере сообщение отправляется прямо по адресу DEST_ADDR:DEST_PORT. В тело сообщения можно включать как двоичные данные, так и ASCII-текст.

Для посылки широковещательных данных требуется два дополнительных действия. Во-первых, требуется перевести сокет в широковещательный режим работы, что делается с помощью вызова функции setsockopt(), имеющей следующий прототип:

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t
optlen);
```

Параметр sockfd – сокет, который переводится в широковещательный режим работы; level (уровень) должен быть обязательно установлен в значение SOL_SOCKET, optname – собственно сама опция сокета – должна принять значение SO_BROADCAST. Два оставшихся параметра указывают на значение параметра (int value = 1 в рассматриваемом случае) и размер значения последнего параметра. Приведем пример:

```
bcfd  = socket( AF_INET, SOCK_DGRAM, 0 );
int one = 1;   ...
setsockopt( bcfd, SOL_SOCKET,  SO_BROADCAST,  (const int *) &one, sizeof(
int ))
```

Следует учесть и целевой адрес рассылки. Лучше всего воспользоваться константой INADDR_BROADCAST:

```
sDestAddr.sin_family = AF_INET;
sDestAddr.sin_len = sizeof(sDestAddr);
sDestAddr.sin_addr.s_addr = htonl(INADDR_BROADCAST);
```

3.2. Прядок выполнения работы. Не используя клиент-серверную архитектуру, реализовать чат в локальной сети с помощью протокола UDP. Различные экземпляры программы, запущенные на разных компьютерах, должны обнаружить друг друга с целью ведения чата. Обнаружение друг друга должно быть реализовано с помощью механизма широковещания.

3.3. Контрольные вопросы. 1. Что такое широковещание? **2.** Для чего предназначено широковещание? **3.** Почему протокол TCP не подходит для широковещания? **4.** Что представляет собой пакет протокола UDP? **5.** Как программно организовать широковещание?

Лабораторная работа №4
ОСВОЕНИЕ ПРОТОКОЛА FTP

Цель работы: формирование практических навыков реализации протоколов высокого уровня.

4.1. Теоретические сведения. ***Протокол FTP.*** Как правило, клиент не оперирует командами FTP непосредственно (рис. 4.1). Эту роль берут на себя интерфейс пользователя FTP и интерпретатор протокола FTP. Интерфейс пользователя может быть представлен в виде графического или консольного приложения (например программа «ftp» в Windows), задачей которого является интерпретация команд пользователя в соответствующие команды протокола FTP и трансформация откликов сервера в удобный формат (табл. 4.1).

Приведем некоторые примеры откликов: 1) однострочные отклики: 200 XXX command successful (команда XXX принята); 500 Syntax error (синтаксическая ошибка, неверная команда); 452 Error writing file (ошибка при записи файла); 2) многострочный отклик: 230 – Welcome to ftp archive; 230 Guest login ok.

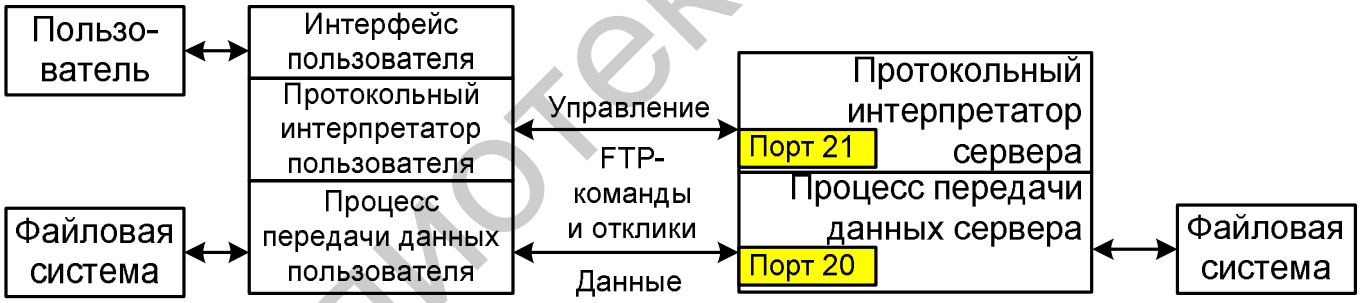

Рис. 4.1. Схема работы протокола FTP

Таблица 4.1

Наиболее употребляемые команды FTP

Команда	Описание
ABOR	Прерывает выполнение предыдущей команды и процесс передачи данных
1	2
CWD pathname	Переходит в директорию pathname на сервере
HELP	Запрашивает список FTP-команд, которые распознаются сервером
LIST filelist	Запрашивает список файлов или каталогов из текущего каталога
PASS password	Пароль для входа на сервер
PASV	Требует, чтобы сервер пассивно открыл свой эфемерный порт и сообщил его номер клиенту, выслав команду PORT

Окончание табл. 4.1

1	2
PORT n1, n2, n3, n4, p1, p2	Содержит IP-адрес (n1.n2.n3.n4) и порт (p1*256+p2) клиента
PWD	Запрашивает имя текущей директории на сервере
QUIT	Вызывает завершение сеанса пользователя на сервере
RETR filename	Запрашивает копию файла с сервера
STOR filename	Записывает файл на сервер
SYST	Запрашивает тип системы сервера
TYPE type	Указывает тип данных файла: A – текстовый (ASCII), I – двоичный (image)
USER username	Имя пользователя для входа на сервер

Следует иметь в виду, что даже если клиент запрашивает несколько файлов с сервера по списку, информационное соединение открывается каждый раз по-новому для каждого файла. При этом стороны действуют по следующей схеме: 1. Открытие информационного соединения инициирует и контролирует клиент, так как именно клиент подает команду (RETR, STOR или LIST), для выполнения которой необходимо создать канал передачи данных. 2. Клиент получает на своем хосте некий номер эфемерного порта для своего конца информационного соединения. 3. Клиент высылает по управляющему соединению команду PORT, сообщая в ней серверу номер своего пассивно открытого эфемерного порта. 4. Получив номер клиентского порта, сервер на своей стороне выполняет активное открытие информационного соединения FTP на этот порт. При этом на своем конце сервер всегда использует порт 20.

Протокол FTP был разработан в университетской среде, когда компьютеров было мало, и они были под управлением UNIX-систем. Поэтому, во-первых, разделителем директорий является символ слэш («/»), и, во-вторых, при выводе списка файлов и директорий используется форматирование стандартной UNIX-утилиты ls. Например:

```
$ ls -la
-rwxr-xr-x  1 dmol wheel   35 2006-10-19 18:21 abiword.sh
-rw-r--r--  1 dmol wheel   90 2007-06-15 20:51 backup-univer.sh
drwxr-xr-x  2 dmol wheel  168 2008-08-21 20:24 dhn
```

Приведем пример FTP-сессии:

```
ftp> open localhost
Связь с alexv-91fddd4e5.
220 TYPSoft FTP Server 1.10 ready...
Пользователь (alexv-91fddd4e5:(none)): anonymous
---> USER anonymous
331 Password required for anonymous.
Password (пароль не отображается)
---> PASS 314@tut.by
230 User anonymous logged in.
ftp> dir
---> PORT 127,0,0,1,4,13
200 Port command successful.
---> LIST
150 Opening data connection for directory list.
-rw-rw-rw-   1 ftp      ftp      1682371 Dec 29  2002 Idef4.pdf
```

```
-rw-rw-rw-   1 ftp      ftp      1435951 Dec 29  2002 Idef5.pdf
-rw-rw-rw-   1 ftp      ftp       468524 Dec 29  2002 Idef9.pdf
226 Transfer complete.
ftp: 474 байта получено за 0,00 с со скоростью 474000,00 Кб/сек.
ftp> get idef4.pdf
---> PORT 127,0,0,1,4,14
200 Port command successful.
---> RETR idef4.pdf
150 Opening data connection for idef4.pdf.
226 Transfer complete.
ftp: 1682371 байт получено за 0,42 (с.) со скоростью 3996,13 (Кб/с.).
ftp> quit
```

Широкое распространение получил сервис *анонимных FTP* (anonymous FTP) –FTP-сервера, предоставляющие всем желающим доступ к своим ресурсам. Обычно именем пользователя при входе на такой сервер служит строка anonymous, а в качестве пароля указывается адрес электронной почты пользователя.

Стандартный ввод-вывод. Существует также стандартный поток ошибок – stderr, поэтому программы должны направлять предупреждающие сообщения и сообщения об ошибках в него, а не в поток stdout. Это позволяет отделять обычные данные, выводимые разного рода служебными сообщениями. Например, стандартный поток вывода можно направить в файл, а сообщения об ошибках по-прежнему отображать на консоли. Запись в поток stderr осуществляется с помощью функции fprintf ():

```
fprintf (stderr, ("Error:   ..."));
```

Все три стандартных потока доступны низкоуровневым функциям ввода-вывода (read(), write() и т. д.) через дескрипторы файлов. В частности, поток stdin имеет дескриптор 0, stdout – 1, а stderr – 2.

При вызове программы иногда требуется одновременно перенаправить потоки вывода и ошибок в файл или в канал. Синтаксис подобной операции зависит от используемого интерпретатора команд. В интерпретаторах семейства Bourne shell (включая bash, который по умолчанию установлен в большинстве дистрибутивов Linux) это делается так:

```
% program > output_file.txt 2>&1 % program 2>&1 | filter
```

Буферизация потока stdout может приводить к неожиданным результатам. Например, в следующем цикле точка не выводится каждую секунду. Вместо этого все символы сначала помещаются в буфер, а затем целая их группа одновременно выводятся на экран, когда буфер оказывается заполненным:

```
while (1) {
printf(".");
sleep(1);}
```

А в нижеследующем цикле точка выводится на экран в момент записи выходного потока:

```
while (1) {
fprintf (stderr, ".");
sleep (1); }
```

Коды завершения системных вызовов. Функция strerror() возвращает строковый эквивалент кода ошибки. Эти строки можно включать в сообщения об ошибках. Объявление функции находится в файле <string. h>.

Функция perror() (объявлена в файле <stdio.h>) записывает сообщение об ошибке непосредственно в поток stderr. Перед собственно сообщением следует размещать строковый префикс, содержащий имя функции или модуля, ставших причиной сбоя.

В следующем фрагменте программы делается попытка открыть файл. Если это не получается, выводится сообщение об ошибке и программа завершает свою работу. Обратите внимание на то, что в случае успеха операции функция open() возвращает дескриптор открытого файла, иначе – 1:

```
fd = open ("inputfile.txt", 0_RDONLY);
  if (fd == -1) { /*открыть файл не удалось. Вывод сообщения об ошибке и
 выход.*/
    fprintf (stderr, "error opening file: %s\n", strerror(errno));
  exit (1); }
```

Один из кодов, с которым приходится сталкиваться наиболее часто, особенно в функциях ввода-вывода – это EINTR. Поэтому в программе следует реализовать обработку этого кода ошибки. Ряд функций, в частности read(), select() и sleep(), требует определенного времени на выполнение. Они называются блокирующими, так как выполнение программы приостанавливается до тех пор, пока функция не завершится. Но если программа, будучи заблокированной, принимает сигнал, функция завершается, не закончив выполнение операции. В данном случае в переменную errno записывается значение EINTR. Обычно в подобной ситуации следует повторно выполнить системный вызов.

4.2. Порядок выполнения работы. Разработать простейший FTP-сервер. При выводе списка файлов не обращать внимания на поля доступа, количество ссылок на файл, владельца файла, группу владельца файла. Должны корректно выводиться дата и время создания файла, его размер. Должна быть реализована возможность смены директорий. В качестве клиентского приложения можно использовать Total Commander для проверки корректности работы программы. Не нужно заботиться о безопасности и реализовывать многозадачность в программе.

4.3. Контрольные вопросы. 1. Для чего предназначен протокол FTP? **2.** Почему в качестве транспортного протокола в протоколе FTP используется TCP-протокол? **3.** Какие преимущества и недостатки того, что информационное соединение открывается каждый раз заново? **4.** Что такое управляющее соединение? **5.** Что такое информационное соединение? **6.** Какие существуют стандартные потоки для ввода-вывода? **7.** Как обрабатывать коды завершения системных вызовов?

Лабораторная работа №5
ОСВОЕНИЕ ПРОТОКОЛА HTTP

Цель работы: формирование практических навыков в реализации протоколов высокого уровня при работе сервера в многопользовательском режиме.

5.1. Теоретические сведения. ***Протокол HTTP.*** Для идентификации объекта доступа по протоколу HTTP используются следующие указатели: 1) универсальный идентификатор ресурса (Uniform Resource Identifier, URI); 2) местонахождение ресурса (Uniform Resource Locator, URL); 3) имя ресурса (Uniform Resource Name, URN).

Принято выделять следующие части данных указателей: 1) схема – идентифицирует тип сервиса, через который можно получить доступ к ресурсу (ftp, http); 2) адрес – идентифицирует хост ресурса (www.microsoft.com); 3) имя или путь доступа – идентифицирует полный путь к ресурсу на указанном хосте (/home/images/image1.gif).

Примером URI является строка http://www.microsoft.com/readme.txt.

Пример URI с параметрами:

http://www.exe.com/bin/run.exe?in=10&str=stop%20and%20go.

Поле Request-URI содержит идентификатор URI-ресурса сервера. Он может быть представлен в абсолютном и относительном форматах. Абсолютный формат содержит все части URI, а относительный – путь к ресурсу на текущем сервере.

В поле HTTP-Version размещается версия протокола HTTP, которую предполагает использовать клиент. С учетом сказанного первая строчка запроса может выглядеть следующим образом: GET /maindoc.html HTTP/1.0.

Секция основного заголовка появляется в запросах только тогда, когда передается тело сообщения (секция тело объекта не является пустой). Это обычно соответствует методам запроса POST или PUT. Секция заголовка запроса позволяет клиенту в запросе передать информацию о себе. Тело объекта передается в запросе в формате, определенном в заголовках. Если тело объекта присутствует, то оно должно отделяться от заголовков пустой строкой. Приведем пример HTTP-запроса:

```
    GET http://tut.by/ HTTP/1.0
    User-Agent: Opera/6.05 (Windows 98; U)  [en]
    Host: tut.by
    Accept: text/html, image/png, image/jpeg, image/gif, */*
    Accept-Language: be,en,ru
    Accept-Charset: windows-1251;q=1.0, utf-8;q=1.0, utf-16;q=1.0, iso-8859-1;q=0.6,
*;q=0.1
    Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
    Cookie2: $Version="1"
    Proxy-Connection: close
```

После получения запроса сервер обрабатывает его и отсылает ответ клиенту. Ответ имеет следующую структуру: 1) поле ответа (Response Line); 2) основной заголовок (General Header); 3) заголовок ответа (Response Header); 4) заголовок объекта (Entity Header); 5) тело объекта (Entity Body). Определено пять классов кодов (табл. 5.1).

Таблица 5.1

Классы кодов

Класс	Описание класса
1xx	Информационный: запрос принят, процесс продолжается
2xx	Успешное завершение: запрос был успешно принят и обработан
3xx	Переназначение: следующее действие должно быть обработано, чтобы завершить запрос
4xx	Ошибка пользователя: запрос содержит неверный синтаксис или не может быть выполнен
5xx	Ошибка сервера: сервер не может выполнить требуемый запрос

Приведем примеры некоторых кодов статуса и соответствующих поясняющих фраз: 1) 200 OK (Норма); 2) 403 Forbidden (Запрет доступа); 3) 404 Not Found (Не найдено). Секции основного заголовка, заголовка объекта и тела объекта идентичны по структуре соответствующим заголовкам HTTP-запроса.

Приведем пример HTTP-ответа:

```
HTTP/1.0 200 OK
Date: Wed, 31 Mar 2004 09:15:05 GMT
Server: Apache/1.3.27 (Unix) mod_ssl/2.8.12 OpenSSL/0.9.6g mod_jk/1.2.0 PHP/4.3.3 mod_perl/1.27
Set-Cookie: brbanner=blueprint1_1080724505; expires=Tue,2 Jan 2007 23:59:59 UTC;
Content-Type: text/html;charset=windows-1251
<HTML><HEAD><TITLE>Kino.br.by</TITLE><META></HTML>
```

Создание многопоточных серверных приложений. Функция pthread_create() создает новый поток. Ей передаются следующие параметры: 1. Указатель на переменную типа pthread_t, в которой сохраняется идентификатор нового потока. 2. Указатель на объект атрибутов потока. Этот объект определяет взаимодействие потока с остальной частью программы. Если задать его равным NULL, поток будет создан со стандартными атрибутами. 3. Указатель на потоковую функцию, которая имеет следующий тип: void* (*)(void*). 4. Значение аргумента потока (тип void*). Данное значение без каких-либо изменений передается потоковой функции.

Функция pthread_create() немедленно завершается, и родительский поток переходит к выполнению инструкции, следующей после вызова функции. Тем временем новый поток начинает выполнять потоковую функцию. ОС планирует работу обоих потоков асинхронно, поэтому программа не должна рассчитывать на какую-то согласованность между ними.

Приведем пример программы, которая создает поток, непрерывно записывающий символы 'x' в стандартный поток ошибок. После вызова

функции pthread_create() основной поток начинает делать то же самое, но вместо символов 'x' печатаются символы 'o':

```
    #include <pthread.h> #include <stdio.h>
/*запись символов 'x' в поток stderr. Параметр не используется. Функция никогда не
завершается.*/
    void* print_xs (void* unused){
    while (1)
    fputc ('x', stderr); return NULL; }
    /* основная программа. */
    int main() {
    pthread_t thread_id;
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    while (l)
    fputc ('o', stderr);
    return 0;           }
```

Запустив программу, можно видеть, что символы 'x' и 'o' чередуются самым непредсказуемым образом. При нормальных обстоятельствах поток завершается одним из двух способов. Один из них – выход из потоковой функции. Возвращаемое ею значение считается значением, передаваемым из потока в программу. Второй способ – вызов специальной функции pthread_exit(). Это может быть сделано как в потоковой функции, так и в любой другой функции, явно или неявно вызываемой из нее. Аргумент функции pthread_exit() является значением, возвращаемым потоком.

Рассмотрим вопрос передачи данных потоку. Одно из решений проблемы заключается в том, чтобы заставить функцию main() дождаться завершения потоков. Нужна лишь функция наподобие wait(), которая работает не с процессами, а с потоками. Такая функция называется pthread_join(). Она принимает два аргумента: идентификатор ожидаемого потока и указатель на переменную void*, в которую будет записано значение, возвращаемое потоком. Если последнее не важно, следует задать в качестве второго аргумента NULL.

Если второй аргумент функции pthread_join () не равен NULL, то в него помещается значение, возвращаемое потоком. Как и потоковый аргумент, это значение имеет тип void*. Если поток возвращает обычное число типа int, его можно свободно привести к типу void*, а затем выполнить обратное преобразование по завершении функции pthread_join ().

Следующая программа в отдельном потоке вычисляет простое число и возвращает его в программу. В это время функция main() может продолжать свои собственные вычисления. Следует отметить, что алгоритм последовательного деления, используемый в функции compute_prime(), весьма неэффективен. Существуют мощные алгоритмы (например «решето Эратосфена»):

```
    #include <pthread.h>
    #include <stdio.h>      /* находим простое число с порядковым номером N, где
                               N - значение, на которое указывает параметр ARG */
    void* compute_prime (void* arg) {
      int candidate = 2;
      int n = *((int*) arg);
      while (1) {
            int factor;
            int  is_prime  =  1;          /*  проверка  простого  числа  путем
последовательного деления. */
            for (factor = 2; factor < candidate; ++factor)
```

```
                if (candidate % factor == 0)  {
                      is_prime = 0; break;    }     /*  это  то  простое  число,
которое нам нужно? */
            if (is_prime) {
                  if (--n == 0)                        /*   возвращаем   найденное
число в программу */
                        return (void*) candidate;      }
            ++candidate;             }
      return NULL;            }
    int main () {
      pthread_t thread;
      int which_prime = 5000;
      int prime;                    /*   запускаем   поток,   вычисляющий   5000-е
простое число */
      pthread_create (kthread, NULL, &compute_prime, &which_prime);
      /* выполняем другие действия. */
      /* дожидаемся завершения потока и принимаем возвращаемое им значение */
      pthread_join (thread, (void*) kprime);               /*       отображаем
вычисленный результат */
      printf("The %dth prime number is %d.\n", which_prime, prime);
      return 0;         }
```

Эти функции удобны для проверки соответствия заданного идентификатора текущему потоку. Например, поток не должен вызывать функцию pthread_join(), чтобы ожидать самого себя (в подобной ситуации возвращается код ошибки EDEADLK). Избежать этой ошибки позволяет следующая проверка:

```
    if (!pthread_equal (pthread_self(), other_thread)) pthread_join (other_thread,
NULL);
```

Рассмотрим атрибуты потоков. Потоковые атрибуты – это механизм настройки поведения отдельных потоков. Вспомните, что функция pthread_create() принимает аргумент, являющийся указателем на объект атрибутов потока. Если этот указатель равен NULL, поток конфигурируется на основании стандартных атрибутов.

Для задания собственных атрибутов потока выполните следующие действия: 1) создайте объект типа pthread_attr_t; 2) вызовите функцию pthread_attr_init(), передав ей указатель на объект, (эта функция присваивает неинициализированным атрибутам стандартные значения); 3) запишите в объект требуемые значения атрибутов; 4) передайте указатель на объект в функцию pthread_create (); 3) вызовите функцию pthread_attr_destroy(), чтобы удалить объект из памяти. Сама переменная pthread_attr_t не удаляется; ее можно проинициализировать повторно с помощью функции pthread_attr_init ().

Один и тот же объект может быть использован для запуска нескольких потоков. Нет необходимости хранить объект после того, как поток был создан.

Чтобы задать статус отсоединения потока, воспользуйтесь функцией pthread_attr_ setdetachstate(). Первый ее аргумент – это указатель на объект атрибутов потока, второй – требуемый статус. Ожидаемые потоки создаются по умолчанию, поэтому в качестве второго аргумента имеет смысл указывать только значение PTHREAD_CREATE_ DETACHED.

Приведем пример программы, которая создает отсоединенный поток, устанавливая соответствующим образом атрибуты потока:

```
#include <pthread.h>
void* thread_function (void* thread_arg) {
/* Тело потоковой функции... */ }
int main () {
  pthread_attr_t attr; pthread_t thread;
  pthread_attr_init (&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)
  pthread_create(&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);
  /*  тело основной программы...   */
  /*  дожидаться  завершения  второго  потока нет  необходимости   */
  return  0;        }
```

Даже если поток был создан ожидаемым, его позднее можно сделать отсоединенным. Для этого нужно вызвать функцию pthread_detach(). Обратное преобразование невозможно.

Рассмотрим вопрос отмены потока. Обычно поток завершается при выходе из потоковой функции или вследствие вызова функции pthread_exit(). Но существует возможность запросить из одного потока уничтожение другого. Это называется отменой, или принудительным завершением, потока.

Чтобы отменить поток, вызовите функцию pthread_cancel(), передав ей идентификатор требуемого потока. Далее можно дождаться завершения потока. Это обязательно следует выполнять с целью освобождения ресурсов, если только поток не является отсоединенным. Отмененный поток возвращает специальное значение PTHREAD_CANCELED.

Чтобы сделать поток асинхронно отменяемым, воспользуйтесь функцией pthread_setcanceltype (). Эта функция влияет на тот поток, в котором она была вызвана. Первый ее аргумент должен быть PTHREAD_CANCEL_ASYNCHRONOUS в случае асинхронных потоков и PTHREAD_CANCEL_DEFERRED – в случае синхронных потоков. Второй аргумент – это указатель на переменную, в которую записывается предыдущее состояние потока.

Вот, как можно сделать поток асинхронным:

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Вот, как можно запретить отмену потока:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

Функция pthread_setcancelstate() позволяет организовывать критические секции. Критической секцией называется участок программы, который должен быть либо выполнен целиком, либо вообще не выполнен. Другими словами, если поток входит в критическую секцию, он во что бы то ни стало должен дойти до ее конца.

Приведем пример функции process_transaction(), осуществляющей данную задумку. Функция запрещает отмену потока до тех пор, пока баланс обоих счетов не будет изменен:

```
    #include <pthread.h>
    #include <stdio.h>
    #include <string.h> /* массив балансов счетов, упорядоченный по номеру счета
*/
    float* account_balances;  /* перевод денежной суммы, равной параметру
DOLLARS,
            со счета FROM_ACCT на счет TO_ACCT. Возвращается 0, если транзакция
```

```
        завершена успешно, или 1, если баланс счета FROM_ACCT слишком мал */
    int process_transaction (int from_acct, int to_acct, float dollars) {
      int old_cancel_state;        /* проверяем баланс на счету FROM_ACCT */
      if (account_balances[from_acct] < dollars)
            return 1;         /* начало критической секции */
      pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
      /* Переводим деньги. */
      account_balances[to_acct] += dollars;
      account_balances[from_acct] -= dollars;        /* конец критической секции
*/
      pthread_setcancelstate (old_cancel_state, NULL);
    return 0;     }
```

Обратите внимание на то, что по окончании критической секции восстанавливается предыдущее состояние потока, а не режим PTHREAD_CANCEL_ENABLE. Это позволит безопасно вызывать функцию process_transaction() из другой критической секции.

После того как ключ создан, каждый поток может назначать ему собственное значение, вызывая функцию pthread_setspecific(). Ее первый аргумент – это ключ, а второй – требуемое значение типа void*. Для чтения потоковых переменных предназначена функция pthread_getspecific(), единственным аргументом которой является ключ.

В следующем листинге показано, как осуществить безопасную запись в файл журнала в многопоточном приложении. Для хранения файлового указателя в функции main() создается ключ, запоминаемый в переменной thread_log_key. Эта переменная является глобальной, поэтому она доступна всем потокам. Когда поток начинает выполнять свою потоковую функцию, он открывает журнальный файл и сохраняет указатель на него в своем ключе. Позднее любой поток может вызвать функцию write_to_thread_log(), чтобы записать сообщение в свой журнальный файл. Эта функция извлекает из области потоковых данных указатель на журнальный файл и помещает в файл требуемое сообщение.

```
    #include <malloc.h>
    #include <pthread.h>
    #include <stdio.h> /*ключ, связывающий указатель журнального файла с каждым
потоком*/
    static pthread_key_t thread_log_key;
    /* запись параметра MESSAGE в журнальный файл текущего потока*/
    void write_to_thread_log (const char* message){
      FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
      fprintf (thread__iog, "%s\n", message);
    }      /*закрытие журнального файла, на который указывает параметр
THREAD_LOG*/
    void close_thread_log (void* thread_log) {
      fclose ((FILE*) thread_log);  }
    void* thread_function (void* args) {
      char thread_log_filename[20];
      FILE* thread_log;  /* создание имени журнального файла для текущего потока */
      sprintf (thread_log_filename, "thread%d.log",
            (int) pthread_self());        /* открытие журнального файла*/
      thread_log = fopen (thread_log_filename, "w");       /* сохранение
указателя файла в области потоковых данных, под ключом thread_log_key */
      pthread_setspecific (thread_log_key, thread_log);
      write_to_thread_log ("Thread starting."); /* далее следует основное тело
потока...*/
      return NULL; }
```

```
    int main () {
      int  i ;
      pthread_t  threads[5]; /* создание ключа, который будет связывать указатели
журнальных файлов с областью данных потока. Функция close_thread_log() закрывает все
файлы*/
      pthread_key_create (&thread_log_key, close_thread_log);    /*создание
потоков*/
      for (i = 0; i < 5;   ++i)
            pthread_create(&(threads[i]), NULL, thread_function, NULL);
      /* ожидание  завершения  всех потоков. */
      for (i  = 0; i < 5;  ++i)
            pthread_join (threads[i],   NULL);
      return  0;  }
```

Для регистрации обработчика следует вызвать функцию pthread_cleanup_push(), передав ей указатель на обработчик и значение его аргумента. Каждому такому вызову должен соответствовать вызов функции pthread_cleanup_pop(), которая отменяет регистрацию обработчика. Для удобства эта функция принимает дополнительный целочисленный флаг. Если он не равен нулю, при отмене регистрации выполняется операция очистки.

В следующем листинге показан фрагмент программы, в котором обработчик очистки применяется для удаления динамического буфера при завершении потока:

```
  #include <malloc.h>
  #include <pthread.h>             /* выделение временного буфера */
  void* allocate_buffer (size_t size) {
      return malloc (size);  }                  /* удаление временного буфера */
  void deallocate_buffer (void* buffer) {
      free (buffer);    }
  void do_some_work () {                  /* выделение временного буфера */
      void* temp_buffer = allocate_buffer (1024); /*регистрация обработчика
очистки для данного буфера. Этот обработчик будет удалять буфер при завершении или
отмене потока*/
      pthread_cleanup_push (deallocate_buffer, temp_buffer); /*выполнение других
действий*/
            /* отмена регистрации обработчика. Поскольку функции передается
      ненулевой аргумент, она выполняет очистку, вызывая функцию
      deallocate_buffer() */
      pthread_cleanup_pop (1);      }
```

Программа, приведенная ниже, иллюстрирует данную методику. Потоковая функция сообщает о своем намерении завершить поток, генерируя исключение ThreadExitException, а не вызывая функцию pthread_exit() явно. Поскольку исключение перехватывается на самом верхнем уровне потоковой функции, все локальные переменные, находящиеся в стеке потока, будут удалены правильно:

```
    #include <pthread.h>
    class  ThreadExitException {
    public:       /* конструктор, принимающий аргумент RETURN_VALUE, в котором
                  содержится возвращаемое потоком значение */
      ThreadExitException (void* return_value):
            thread_retum_value_(return_value) { } /* реальное завершение потока.
В программу возвращается      значение, переданное конструктору */
      void* DoThreadExit () {
            pthread_exit (thread_return_value_);      }
    private:          /* значение, возвращаемое в программу при завершении
потока*/
      void* thread_return_value_; };
    void do_some_work() {
```

```
    while (1) {                    /* здесь выполняются основные действия... */
    if (should_exit_thread_immediately ())
          throw ThreadExitException (/* поток возвращает */ NULL);    } }
  void* thread_function (void*){
    try {
          do_some_work () ;
    } catch (ThreadExitException ex) { /* возникла необходимость завершить
поток*/
          ex.DoThreadExit ();
    } return NULL;  }
```

5.2. Порядок выполнения работы. Разработать простейший HTTP-сервер. Следует учесть, что после запроса страницы веб-браузеры обычно запрашивают все относящиеся к документу файлы (такие как изображения, таблицы стилей и т. п.), поэтому сервер должен поддерживать многозадачность.

5.3. Контрольные вопросы. 1. Назначение протокола HTTP. **2**. В чем различие протоколов HTTP и FTP? **3.** В чем отличие синхронных потоков от асинхронных? **4.** Приведите пример необходимости использования потоковых данных.

Лабораторная работа №6
ОСВОЕНИЕ ШЛЮЗОВОГО ПРОТОКОЛА CGI

Цель работы: формирование практических навыков в реализации интерфейса серверного вызова программ (CGI).

6.1. Теоретические сведения. ***Интерфейс CGI.*** На файловый дескриптор стандартного потока ввода посылается CONTENT_LENGTH байт. Также сервер передает шлюзу CONTENT_TYPE (тип передаваемых данных). Сервер не обязан посылать символ конца файла после отсылки CONTENT_LENGTH байт данных и после того, как шлюз их прочитает.

В качестве примера возьмем результат работы формы с методом POST (METHOD="POST"). Пусть получено 7 байтов, закодированных примерно так: a=b&b=c. В этом случае сервер установит значение CONTENT_LENGTH равным 7 и CONTENT_TYPE в application/x-www-form-urlencoded. Первым символом в стандартном потоке ввода для шлюза будет «a», за которым будет следовать остаток закодированной строки.

Шлюз в командной строке в качестве первого параметра получает от сервера остаток URL после имени шлюза (первый параметр будет пуст, если присутствовало только имя шлюза) и список ключевых слов – в качестве остатка командной строки для скрипта поиска или чередующиеся имена полей формы с добавленным знаком равенства (на четных позициях) и соответствующие значения полей (на нечетных позициях).

Запросы оператора FORM обрабатываются таким образом, что каждый параметр, отвечающий за имя поля, оканчивается знаком равенства, а остаток представляет собой значение этого параметра. Если присутствует что-либо после имени скрипта (шлюза), то эта информация передается в качестве первого параметра. Иначе первый параметр будет пуст.

Например, по URI

/htbin/foo/x/y/z?name1=value1&name2=value2

Скрипт вызывается так : /.../foo /x/y/z name1= value1 name2= value2

А по URL /htbin/foo?name1=value1&name2=value2

/.../foo " name1= value1 name2= value2

Опишем переменные окружения CGI. Переменные окружения не являются специфичными по типу запросов и устанавливаются для всех запросов – SERVER_SOFTWARE: 1. Название и версия информационного сервера, который отвечает на запрос (и запускает шлюз). Формат: имя/версия – SERVER_NAME. 2. Имя хоста, на котором запущен сервер, DNS-имя или IP-адрес в том виде, в котором он представлен в URL – GATEWAY_INTERFACE. 3. Версия CGI-спецификации на тот момент, когда компилировался сервер. Формат: CGI/версия. Переменные окружения являются специфичными для разных запросов и заполняются перед вызовом шлюза – SERVER_PROTOCOL: 1. Имя и версия информационного протокола, в котором пришел запрос. Формат: протокол/версия – SERVER_PORT. 2. Номер порта, на который был послан запрос – REQUEST_METHOD. 3. Метод, который был использован для запроса. Для HTTP, это GET, HEAD, POST, и т. д. – PATH_INFO.

Вывод шлюза начинается с маленького заголовка. Он содержит текстовые строки в том же формате, что и формат строк http-заголовке, и завершается пустой строкой, содержащей только символ перевода строки или CR/LF.

Любые строки заголовка, не являющиеся директивами сервера, посылаются непосредственно клиенту. В настоящий момент CGI-спецификация определяет три директивы сервера:

```
Content-type
MIME тип возвращаемого документа.
Location
```

Поле Location используется в случае, когда необходимо указать серверу, что возвращается не сам документ, а ссылка на него.

Если аргументом является URL, то сервер передаст клиенту указание на перенаправление запроса. Если аргумент представляет собой виртуальный путь, сервер вернет клиенту заданный этим путем документ, как будто клиент запрашивал его непосредственно.

```
Status
```

Эта директива используется для задания серверу HTTP/1.0 строки-статуса, которая будет послана клиенту. Формат: nnn xxxxx, где nnn – 3-цифровой статус-код и xxxxx строка причины, такая как «Forbidden» (Запрещено).

Приведем несколько примеров. Рассмотрим шлюз, который в некоторых случаях, должен выдать документ /path/doc.txt с данного сервера, как если бы он был непосредственно востребован клиентом через http://server:port/path/doc.txt. В это случае вывод шлюза будет таков:

```
--- начало вывода ---
Location: /path/doc.txt
--- конец вывода ---
```

Пусть имеется некоторый текстовый конвертер в HTML. Когда он оканчивает свою работу, он должен произвести следующий вывод в

стандартный выходной поток:

```
--- начало вывода ---
Content-type: text/html
--- конец вывода ---
```

Типы MIME. Приведем примеры часто используемых типов MIME. Тип application обычно указывает на то, что содержимое должно быть обработано специфическим программным обеспечением. Подтипы: 1) application/javascript – содержимое является javascript файлом; 2) application/octet-stream – содержимое является произвольным потоком байтов; 3) application/ogg – содержимое является аудиофайлом в формате ogg; 4) application/xhtml+xml – содержимое – это веб-страничка в формате xhtml; 5) application/zip – содержимое представляет собой zip-архив. Тип audio: 1) audio/mpeg; 2) audio/x-wav. Тип image: 1) image/gif; 2) image/png; 3) image/jpeg. Тип text: 1) text/css; 2) text/html; 3) text/xml; 4) text/plain.

Переменные окружения или среда выполнения. Приведем пример программы, осуществляющей итерацию по всем переменным среды.

```
 #include <stdio.h>              /* Массив ENVIRON содержит среду выполнения
*/
extern char** environ;
int main() {
  char** var;
  for (var = environ; *var != NULL; ++var)
        printf ("%s\n", *var);
  return 0; }
```

Программирование процессов. У любого процесса имеется родительский процесс (за исключением специального корневого процесса ядра). Таким образом, все процессы Linux организованы в виде древовидной иерархии, на вершине которой находится корневой процесс. К атрибутам процесса относится идентификатор его предка (PPID, parent process identifier). Работая с идентификаторами процессов в программах, написанных на языках C и C++, следует объявлять соответствующие переменные как переменные, имеющие тип pid_t (определен в файле <sys/types.h>). Программа может узнать идентификатор своего собственного процесса с помощью системного вызова getpid(), а идентификатор своего родительского процесса – с помощью вызова getppid(). В следующем листинге показано, как это сделать:

```
#include<stdio.h>
#include<unistd.h>
int main()    {
  printf("The process ID is %d\n", (int) getpid());
  printf("The parent process ID is %d\n", (int) getppid ());
  return 0;   }
```

Функция system() определена в стандартной библиотеке языка C и позволяет вызывать из программы системную команду, как если бы она была набрана в командной строке. Эта функция запускает стандартный интерпретатор и передает ему команду на выполнение. Например, программа, представленная в листинге, вызывает команду ls -1 /, отображающую содержимое корневого каталога:

```
    #include <stdlib.h>
int main()    {
  int return_value;
        return_value = system ("ls -1 /");
        return return_value;    }
```

Функция system() возвращает код завершения указанной команды. Если интерпретатор не может быть запущен, возвращается значение 127, а в случае возникновения других ошибок – минус 1. Вызывая функцию fork(), программа создает свой дубликат, называемый дочерним процессом. Родительский процесс продолжает выполнять программу с той точки, где была вызвана функция fork(). То же самое делает и дочерний процесс.

В следующем листинге приведен пример ветвления программы с помощью функции fork(). Учтите, что первая часть инструкции if выполняется только в родительском процессе, тогда как ветвь else – только в дочернем:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()   {
        pid_t child_pid;
  printf  ("The main program process ID is %d\n",   (int)  getpid  ());
  child_pid = fork ();
        if  (child_pid != 0){
        printf ("This is the parent process, with ID %d\n", (int)   getpid
());
        printf  ("The child's process ID is %d\n",   (int)  child_pid);
  }else
        printf   ("This is the child process, with ID %d\n", (int)   getpid
());
  return 0;   }
```

Функции, в названии которых присутствует суффикс 'v' (execv(), execvp() и execve()), принимают список аргументов программы в виде массива строковых указателей, оканчивающегося NULL-указателем. Функции с суффиксом 'l' (execl(), execlp() и execle()) принимают список аргументов переменного размера. Функции, в названии которых присутствует суффикс 'e' (execve() и execle()), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид «ПЕРЕМЕННАЯ=значение».

Программа, приведенная в следующем листинге, отображает содержимое корневого каталога с помощью команды ls. Но на этот раз команда ls вызывается не из интерпретатора, а напрямую; ей передаются аргументы – 1 и /:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>       /*  запуск  дочернего  процесса  в  виде  новой
программы. Параметр PROGRAM – это имя вызываемой программы; ее поиск будет
осуществляться в каталогах,  определяемых переменной среды PATH. Параметр
ARG_LIST – это список строковых аргументов, передаваемых программе (должен
оканчиваться указателем NULL). Функция возвращает идентификатор порожденного
процесса.  */
int spawn (char* program, char** arg_list) {
  pid_t child_pid;                    /*  создание  копии  текущего  процесса
*/
```

```
  child_pid = fork ();
  if  (child_pid != 0)                    /* это родительский процесс  */
  return child_pid;
  else {                                  /* выполнение указанной программы */
  execvp (program, arg_list); /*функция execvp() возвращает значение только в случае
ошибки*/
        fprintf  (stderr,   "an error occurred in execvp\n");
        abort  ();  }     }
int main () {             /* список аргументов, передаваемых команде ls */
  char* arg_list[] = {
        "ls",                             /* argv[0]  -- имя программы  */
        "-l",
        "/",
        NULL  /* список аргументов должен оканчиваться указателем NULL   */
  };               /* порождаем дочерний процесс, который выполняет команду
ls. Игнорируем возвращаемый идентификатор дочернего процесса. */
  spawn ("ls", arg_list);
  printf ("done with main program\n");
  return 0;  }
```

Взаимодействие процессов посредством каналов. В интерпретаторе команд канал создается оператором |. Например, приведенная команда обеспечивает запуск интерпретатором два дочерних процесса: один – для программы ls, а второй – для программы less: % ls | less.

Канал создается с помощью функции pipe(). Ей необходимо передать массив из двух целых чисел. В элементе с индексом 0 функция сохраняет дескриптор файла, соответствующего выходному концу канала, а в элементе с индексом 1 сохраняется дескриптор файла, соответствующего входному концу канала. Рассмотрим следующий фрагмент программы:

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe  (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[l];
```

В программе, приведенной в следующем листинге, родительский процесс записывает в канал строку, а дочерний процесс читает ее. С помощью функции fdopen() файловые дескрипторы приводятся к типу FILE*, благодаря чему появляется возможность использовать функции ввода-вывода: printf () и fgets ().

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>   /* запись указанного числа копий (COUNT) сообщения
(MESSAGE)
                         в поток (STREAM) с паузой между каждой операцией
*/ void writer (const char* message, int count, FILE* stream) {
  for (; count > 0; --count) { /*запись сообщения в поток с "выталкиванием" из буфера*/
        fprintf  (stream,   "%s\n", message);
        fflush (stream);                  /* небольшая пауза  */
        sleep (1) ;       } }   /*чтение  строк  из  потока,  пока  он  не
опустеет */
void reader (FILE* stream) {
  char buffer[1024];              /*чтение данных, пока не будет обнаружен конец
                                  потока.  Функция  fgets()  завершается,  когда
встречает
                                  символ новой строки или признак конца файла*/
  while (!feof  (stream) &&  !ferror (stream)
        && fgets  (buffer,  sizeof  (buffer),  stream) != NULL)
        fputs  (buffer,  stdout); }
```

```
int main () {
  int fds[2]; pid_t pid;      /*создание канала. Дескрипторы обоих концов канала
                              помещаются в массив FDS*/
  pipe  (fds);                          /* порождение дочернего процесса  */
  pid = fork();
  if  (pid == (pid_t) 0){
        FILE* stream;     /*  это дочерний процесс. Закрываем копию входного конца
канала */
        close (fds[l]); /*Приводим дескриптор выходного конца канала к типу
FILE*
                              и читаем данные из канала*/
        stream = fdopen (fds[0], "r");
        reader (stream) ;
        close (fds[0]);
  }else {                       /* это родительский процесс  */
        FILE* stream;           /* закрываем копию выходного конца канала */
        close  (fds[0]);        /* приводим дескриптор входного конца канала
                                к типу FILE* и записываем данные в канал */
        stream = fdopen (fds[l],   "w");
        writer ("Hello, world.",  5,  stream);
        close (fds[l]);   }
  return 0; }
```

Часто требуется создать дочерний процесс и сделать один из концов его канала стандартным входным или выходным потоком. В этом случае на помощь приходит функция dup2(), которая делает один файловый дескриптор равным другому. Вот как, например, можно связать стандартный входной поток с файлом fd:

```
dup2 (fd,  STDIN_FILENO);
```

Программа, представленная в следующем листинге, с помощью функции dup2() соединяет выходной конец канала со входом команды sort. После создания канала программа «делится» функцией fork() на два процесса. Родительский процесс записывает в канал различные строки, а дочерний процесс соединяет выходной конец канала со своим входным потоком, после чего запускает команду sort:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main () {
  int fds[2];
  pid_t pid; /*создание канала. Дескрипторы обоих концов канала помещаются в массив
FDS.*/
  pipe(fds);                            /* создание дочернего процесса  */
  pid = fork ();
  if (pid == (pid_t)  0){  /*это дочерний процесс. Закрываем копию входного конца
канала*/
        close  (fds[1]);  /*соединяем выходной конец канала со стандартным входным
потоком*/
        dup2  (fds[0],    STDIN_FILENO); /*замещаем дочерний процесс программой
sort*/
        execlp ("sort",   "sort",  0);
  }else {                       /* это родительский процесс  */
        FILE* stream;           /*  закрываем копию выходного конца канала
*/
        close(fds[0]);          /*  приводим дескриптор входного конца
канала  к
```

```
                                типу FILE* и записываем данные в канал  */
        stream = fdopen (fds[1],  "w");
        fprintf (stream, "This is a test.\n");
        fprintf (stream, "Hello, world.\n");
        fprintf (stream, "My dog has fleas.\n");
        fprintf (stream, "This program is great.\n");
        fprintf (stream,  "One fish, two fish.\n");
        fflush (stream);
        close (fds[1]);         /* дожидаемся завершения дочернего процесса
*/
        waitpid (pid, NULL,  0);           }
  return 0;   }
```

Каналы часто используются для передачи данных программе, выполняющейся как подпроцесс (или приема данных от нее). Специально для этих целей предназначены функции popen() и pclose(), устраняющие необходимость в вызове функций pipe(), dup2(), exec() и fdopen().

```
#include <stdio.h>
#include <unistd.h>
int main () {
  FILE* stream = popen  ("sort",   "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish,  two fish.\n");
  return pclose (stream); }
```

Функция pclose() закрывает поток, указатель на который был возвращен функцией popen(), и дожидается завершения дочернего процесса.

6.2. Порядок выполнения работы. Доработать HTTP-сервер таким образом, чтобы он минимальным образом поддерживал интерфейс CGI. В качестве CGI-программы использовать php-интерпретатор. Попробовать «вызвать из браузера» какую-нибудь консольную системную программу, например route.

6.3. Контрольные вопросы. 1. Для чего предназначен интерфейс CGI? **2.** Объясните основную идею взаимодействия веб-сервера и CGI-программы. **3.** Для чего предназначены типы MIME? Приведите примеры. **4.** Как осуществляется доступ к переменным окружения? **5.** Как запустить процесс из текущей программы? **6**. Как осуществляется коммуникация между процессами посредством каналов?

Лабораторная работа №7
ОСВОЕНИЕ ПРОТОКОЛА ПЕРЕСЫЛКИ СООБЩЕНИЙ SMTP

Цель работы: формирование практических навыков в использовании протокола SMTP.

7.1. Теоретические сведения. В стеке протоколов TCP/IP протокол SMTP использует транспорт TCP. Номер порта, зарезервированного для сервера SMTP, – 25. Подобно протоколу FTP протокол SMTP работает по принципу «запрос – ответ»: клиент посылает команды серверу, сервер возвращает отклики клиенту (табл. 7.1).

Команды клиента и отклики (табл. 7.2) сервера передаются по управляющему соединению как строки текста. Каждая команда или отклик

заканчивается парой байт CR и LF (возврат каретки и перевод строки). Общая длина команды не должна превышать 1000 байтов. Для того чтобы программа SMTP-сервера была работоспособна, она должна понимать следующий минимум команд: HELO, MAIL, RCPT, DATA, RSET, NOOP, QUIT.

Для инициализации канала обмена почтой и его закрытия используются команды HELO и QUIT соответственно. Первая команда сеанса – HELO.[1]

Наиболее распространенным действием, выполняемым по протоколу SMTP, является отправка почтового сообщения. Отправка начинается по команде MAIL, идентифицирующей отправителя: MAIL FROM: inform@bsuir.unibel.by[2].

Таблица 7.1

Наиболее употребляемые команды SMTP

Команда	Описание
HELO domain	Открыть сессию взаимодействия по протоколу SMTP. domain – доменное имя хоста отправителя
MAIL FROM:<reverse-path>	Сообщить адрес отправителя (<reverse-path>). Обязательная команда перед отправкой сообщения
RCPT TO:<forward-path>	Сообщить адрес получателя (<forward-path>). Обязательная команда, которую выдают после MAIL, но перед DATA
DATA	Начать передачу почтового сообщения. Признак конца сообщения – строка, состоящая из точки («.»)
RSET	Сброс текущей почтовой транзакции
VRFY user	Получить информацию о пользователе user
EXPN userlist	Получить информацию о пользователях, зарегистрированных в качестве получателей корреспонденции
HELP [command][3]	Краткая справка по командам протокола
NOOP	Нет операции
QUIT	Завершить сессию
TURN	Поменяться местами серверу и клиенту

Таблица 7.2

Расшифровка значений отклика сервера

Значение первой цифры отклика		Значения второй цифры отклика	
Код	Назначение	Код	Назначение
1	2	3	4
1yz	Позитивный промежуточный отклик. Команда воспринята, отправитель должен послать следующую команду	x0z	Синтаксис – эти отклики относятся к синтаксическим ошибкам или к командам, синтаксически корректным, но примененным неправильно
2yz	Позитивное подтверждение завершения операции. Можно посылать следующий запрос	x1z	Информация – относится к командам, которые запрашивают информацию, например статусную или справочную

1. Если первой командой сеанса будет команда HELO, сервер выдаст список команд, составляющих дополнение списка стандартных команд протокола SMTP.
2. Некоторые почтовые серверы требуют написания электронного адреса в угловых скобках < >.
3. Аргументы в квадратных скобках являются необязательными.

Окончание табл. 7.2

1	2	3	4
3yz	Позитивный промежуточный отклик, сходный с 1yz. Используется в случае групповых команд	x2z	Соединения – относится к телеком-муникационному каналу
4yz	Временный негативный отклик. Команда не исполнена, но характер ошибки временный и выполнение процедуры может быть позже повторено	x3z	Пока не определен
5yz	Окончательный негативный отклик. Команда не воспринята, запрошенная операция не выполнена и не будет выполнена	x4z	Пока не определен
		x5z	Почтовая система – эти отклики индицируют статус получателя или отправителя почты.

Следующей командой определяется адрес получателя: RCPT TO: 314@tut.by.

Приведем листинг обмена сообщениями при отправке почты. Команды клиента выделены жирным шрифтом, отклики сервера – курсивом:

```
220 TUT.BY ESMTP Server at speedy.tutby.com is ready to rock!
HELO mail.tut.by
250 tut.by your name is not mail.tut.by
MAIL FROM: inform@bsuir.unibel.by
250 inform@bsuir.unibel.by sender accepted
RCPT TO: 314@tut.by
250 314@tut.by will leave the Internet
DATA
354 Enter mail, end with "." on a line by itself
From: inform@bsuir.unibel.by
To: 314@tut.by
Subject: Test
This is a test
Second line.
250 52963409 message accepted for delivery
QUIT
221 tut.by TUT.BY SMTP Server at speedy.tutby.com is closing connection
```

Достаточно часто протокол SMTP выполняет задачи перенаправления почтовых сообщений (forwarding). Если получатель не найден, но известно его местоположение, то сервер может выдать сообщение:

```
251 User not local; will forward to <user@domain.domain>
```

Если сервер может сделать только предположение о дальнейшей рассылке, то ответ будет несколько иным:

```
551 User not local; please try <user@host.domain>
```

Еще одной задачей протокола SMTP является задача верификации и расширения почтовых адресов. В ней используются команды VRFY и EXPN. По команде VRFY сервер подтверждает наличие или отсутствие указанного пользователя:

```
VRFY paul
250-Paul Khramtsov<paul@quest.polyn.kiae.su>
```

Используя команду EXPN, можно получить список местных пользователей:

```
EXPN Example-People
250-Paul Khramtsov<paul@quest.polyn.kiae.su>
250-Vladimir Drach-Gorkunov<vovka@quest.polyn.kiae.su>
```

Аргументом команд EXPN служит строка-идентификатор списка адресатов домена. Это может быть имя файла со списком или какой-либо другой идентификатор списка, определенный в системе.

Рассмотрим некоторые другие команды, используемые в SMTP. Команда TURN используется для того, чтобы поменять местами функции взаимодействующих почтовых агентов. Агент-отправитель становится получателем (после того как она выдаст команду TURN и получит отклик 250), а агент-получатель – отправителем. Если агент не хочет или не может поменять свою функцию, она пошлет отклик 502. Команда RSET прерывает текущую процедуру отправки почтового сообщения. Все буферы и таблицы очищаются, получатель должен послать отклик 250 OK. Команда HELP вынуждает сервер послать справочную информацию отправителю. Команда NOOP не оказывает влияния на какие-либо параметры или результаты предшествующих команд, она только вынуждает получателя послать отклик 250 OK. Эта команда может использоваться для проверки работоспособности TCP-соединения.

7.2. Порядок выполнения работы. Реализовать простейшую программу-клиент для отсылки электронной почты.

7.3. Контрольные вопросы. 1. Принцип работы протокола SMTP. **2.** Какой протокол используется в качестве транспортного для протокола SMTP? **3.** Для чего нужна команда HELO? **4.** Возможно ли посредством протокола SMTP отправить письмо, в теле которого имеется строка, с единственным символом – точкой?

ЛИТЕРАТУРА

1. Таненбаум, Э. Компьютерные сети / Э. Таненбаум. – 4-е изд. – СПб. : Питер, 2008. – 999 с.

2. Таненбаум, Э. Современные операционные системы / Э. Таненбаум. – 2-е изд. – СПб. : Питер, 2007. – 1038 с.

3. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум. – 5-е изд. – СПб. : Питер, 2007. – 848 с.

4. Олифер, В. Г. Компьютерные сети. Принципы, технологии, протоколы / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2005. – 864 с.

Св. план 2010, поз. 17

Учебное издание

ИНТЕРФЕЙС В СИСТЕМАХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Лабораторный практикум
для студентов специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий»
всех форм обучения

Составители:
Байдаков Иван Владимирович
Осипович Виталий Семенович
Яшин Константин Дмитриевич

Редактор Л. А. Шичко
Корректор Е. Н. Батурчик
Компьютерная верстка А. В. Тюхай

Подписано в печать | Формат 60x84 1/16. | Бумага офсетная.
Гарнитура «Таймс». | Отпечатано на ризографе. | Усл. печ. л.
Уч.-изд. л. 2,0. | Тираж 70 экз. | Заказ 186.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 30.04.2009.
220013, Минск, П. Бровки, 6