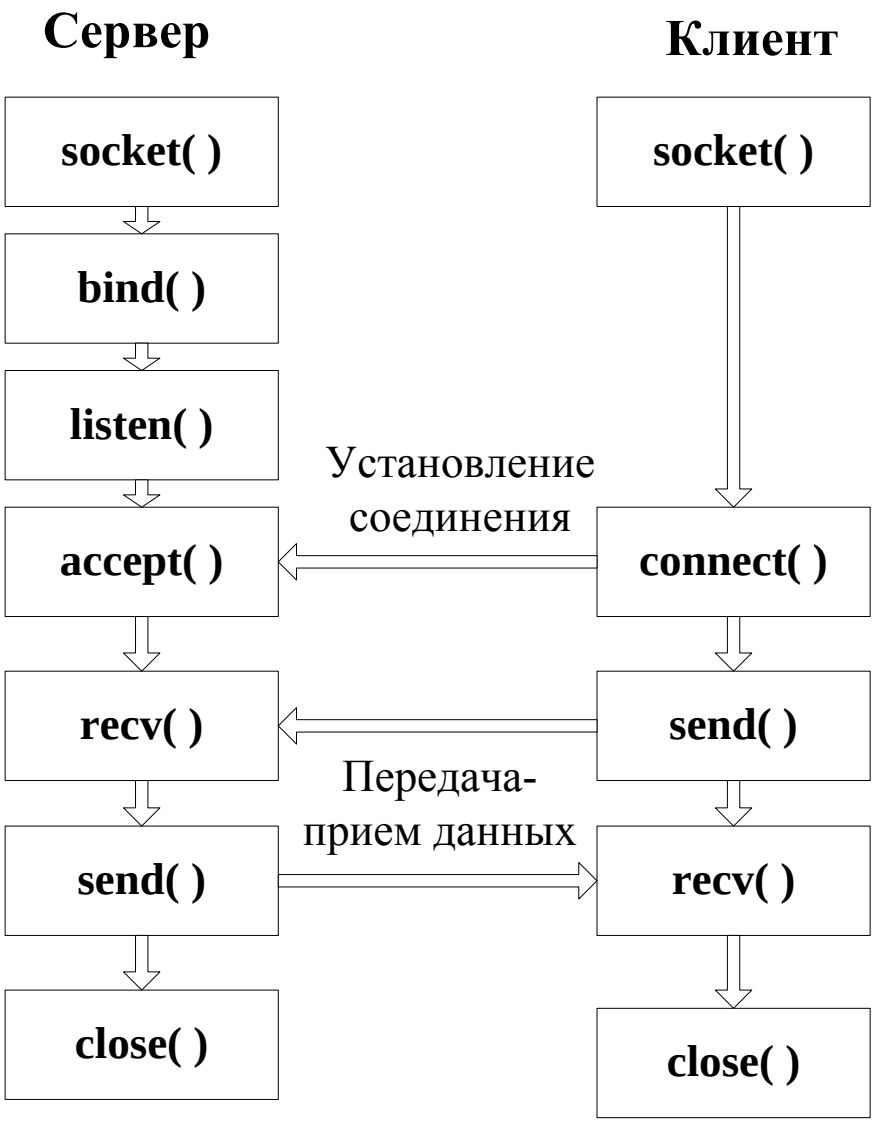
Программа типа клиент-сервер для TCP



API Socket

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

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

Аргумент *domain* определяет правила именования гнезда и формат адреса, используемые в протоколе. Широко применяются такие домены, как *AF_UNIX* (домен UNIX) и *AF_NET* (Internet-домен).
Аргумент *type* задает тип гнезда.
Аргумент *protocol* указывает конкретный протокол, который следует использовать с данным гнездом. Как правило, оно устанавливается в 0, и операционная система сама выбирает для указанного типа гнезда соответствующий протокол. Например, для типа гнезда *SOCK_STREAM* третьему аргументу по умолчанию выбирается значение TCP.

В случае успешного выполнения рассматриваемая функция возвращает целочисленный дескриптор гнезда, а в случае неудачи возвращает -1.

int close (int sid); Закрытие сокета *sid* .

API Socket

int listen (int sid, int size);

Эта функция вызывается серверным процессом для создания гнезда, ориентированного на установление соединения.
Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.*
Аргумент *size* задает максимальное число запросов на установление соединения, которые могут быть поставлены в очередь к данному гнезду.
При успешном выполнении эта функция возвращает 0, а в случае неудачи возвращает -1.

API Socket

int accept (int sid, struct sockaddr addr_p, int len_p);*

Эта функция вызывается в серверном процессе для установления соединения с клиентским гнездом (которое делает запрос на установление соединения посредством вызова функции *connect).*

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.*

Аргумент *addr_p* — это указатель на адрес объекта типа *struct sockaddr*; в нем хранится имя клиентского гнезда, с которым устанавливает соединение серверное гнездо (тип *struct sockaddr_in*).

Аргумент *len_p* изначально устанавливается равным максимальному размеру объекта, указанному аргументом *addr_p.* При возврате он содержит размер имени клиентского гнезда, на которое указывает аргумент *addr_p.*
Если аргумент *addr_p* или аргумент *len_p* имеет значение NULL, эта функция не передает имя клиентского гнезда обратно в вызывающий процесс.

В случае неудачи рассматриваемая функция возвращает -1. В противном случае она возвращает дескриптор нового гнезда, с помощью которого серверный процесс может взаимодействовать исключительно с данным клиентом.

API Socket

int connect (int sid, struct sockaddr addr_p, int len);*

Эта функция вызывается в клиентском процессе для установления соединения с серверным гнездом.
Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.* Второй аргумент *addr_p* — это указатель на адрес объекта типа *struct sockaddr*. Для домена Internet структура адреса сокета имеет тип *struct sockaddr_in.* Должно быть выполнено соответствующее преобразование типа.

Аргумент *len* задает размер объекта (в байтах), на который указывает аргумент *addr_p.*

Если *sid* обозначает потоковое гнездо, то между клиентским и серверным гнездами устанавливается соединение с использованием виртуального канала. Потоковое гнездо клиента может соединяться с гнездом сервера только один раз. Если *sid* обозначает датаграммное гнездо, то для всех последующих вызовов функции *send,* осуществляемых через это гнездо, устанавливается адрес по умолчанию. Датаграммное гнездо может соединяться с гнездом сервера многократно, изменяя установленные по умолчанию адреса. Путем соединения с гнездом, имеющим NULL-адрес, датаграммные гнезда могут разорвать соединение.

При успешном выполнении эта функция возвращает 0, а в случае неудачи — 1.

API Socket

int send (int sid, const char* buf, int len, int flag);

Эта функция передает содержащееся в аргументе *buf* сообщение длиной *len* байтов в гнездо, заданное аргументом *sid* и соединенное с данным гнездом.
Аргументу *flag* обычно присваивается значение 0, но он может иметь и значение MSG_OOB. В этом случае сообщение, содержащееся в *buf* должно быть передано как высокоприоритетное (out-of-band message).
В случае неудачи эта функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

int sendto (int sid, const char* buf, int len, int flag, struct sockaddr* addr_p, int* len_p);

Эта функция делает то же самое, что и API *send*, только вызывающий процесс указывает также адрес гнезда-получателя (в аргументах *addr_p и len _p).*
Аргументы *sid, buf, len* и *flag* — те же самые, что в API *send.*

Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-получателя (тип *struct sockaddr_in*). Аргумент *len_p* содержит число байтов в объекте, на который указывает аргумент *addr_p.*
В случае неудачи данная функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

API Socket

int recv (int sid, char* buf, int len, int flag);

Эта функция принимает сообщение через гнездо, указанное в аргументе *sid.* Принятое сообщение копируется в буфер *buf*, а максимальный размер *buf* задается аргументом *len.* Если в аргументе *flag* указан флаг MSG_OOB, то приему подлежит высокоприоритетное сообщение. В противном случае ожидается обычное сообщение. Кроме того, в аргументе *flag* может быть указан флаг MSG_PEEK, означающий, что процесс желает "взглянуть" на полученное сообщение, но не собирается удалять его из потокового гнезда. Такой процесс может повторно вызвать функцию *recv* для приема сообщения позже.

В случае неудачи функция *recv* возвращает -1; в случае успешного выполнения возвращается число байтов данных, записанных в буфер *buf.*

int recvfrom (int sid, const char* buf, int len, int flag, struct sockaddr* addr_p, int* len_p);

Эта функция делает то же самое, что и API *recv,* только при ее вызове задаются аргументы *addr_p* и *len_p,* позволяющие узнать имя гнезда-отправителя.
Аргументы *sid, buf len* и *flag* — те же самые, что в API *recv.*
Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-отправителя (тип *struct sockaddr_in*).
Аргумент *len_p* сообщает число байтов в объекте, на который указывая аргумент *addr_p.*

API Socket (Пример цикл для recv)

Пример:

```
#define BLEN 120            /* Длина используемого буфера */
        char *req = "request of  some sort";
        char buf[BLEN] ;            /* Буфер для ответа */
        char *bptr;                 /* Указатель на буфер */
        int n;                      /* Количество считанных байтов */
        int buflen;                 /* Место, оставшееся в буфере */
        bptr = buf;
        buflen = BLEN;

 send(s, req, strlen(req), 0);   /* Отправить запрос */
        /* Получить ответ (может состоять из нескольких фрагментов) */

 while((  n = recv(s, bptr, buflen, 0)  ) > 0) {
             bptr+= n;
             buflen–= n;
        }
```

Единственный вызов функции send, но предусматривает повторное выполнение вызовов функции **recv**. До тех пор, пока вызовы функции r**ecv** возвращают данные, в коде уменьшается счетчик пространства, доступного в буфере, а указатель буфера продвигается вслед за полученными данными. Такая итерация необходима даже если приложение на другом конце соединения передает каждый раз только небольшой объем данных, поскольку протокол TCP не гарантирует доставку их в виде таких же фрагментов, какие были отправлены.

API Socket

int shutdown (int sid, int mode);

Данная функция закрывает соединение между серверным и клиентским гнездами.
Аргумент *sid* — это дескриптор гнезда, возвращенный функцией *socket* .
Аргумент *mode* задает режим закрытия.

Режим	**Пояснение**
0	Закрывает гнездо для чтения. При попытке продолжить чтение будут возвращаться нулевые байты (EOF)
1	Закрывает гнездо для записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения, -1
2	Закрывает гнездо для чтения и записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения -1, а при продолжении чтения будет возвращаться нулевое значение (EOF

В случае неудачи данная функция возвращает -1, а в случае успешного выполнения — 0.

API Socket (вспомогательные функции)
gethostbyname

```
struct hostent
{
        char   *h name;                 /* Доменное имя хоста */
        char **h_aliases;               /* Псевдонимы */
        int       h_addrtype;           /* Тип адреса */
        int       h_length;             /* Длина адреса */
        char **h_addr_list;             /* Список адресов */
        #define h_addr h_addr_list[0]
}
```

Пример:

```
struct hostent *phe;
char *name = "csc.neic.nsk.su";

    if  (phe = gethostbyname(name))
        {
            / * Теперь IP-адрес - в поле phe–>h_addr */
         }
     else          /* Ошибка в имени - обработать ошибку */ }
```

API Socket (getservbyname)

```
struct servent
{
    char     *s_name;            /*  Стандартное имя службы */
    char   **s_aliases;        /*  Псевдонимы */
    int         s_port;             /*  Порт службы */
    char     *s_proto;           /*  Используемый протокол */
}
```

Пример:

Пусть необходимо найти официально назначенный номер порта протокола для SMTP.

```
struct servent *pse;

 if  ( pse = getservbyname("smtp",  "tcp"))
     {
            /* Теперь номер порта - в поле pse–>s_port */
      }
  else      /* Возникла ошибка - обработать ошибку */
```

API Socket (getprotobyname)

```
struct protoent
{
        char   *p_name;          /* Стандартное имя протокола */
        char **p_aliases;        /* Список допустимых псевдонимов */
        int       p_proto;        /* Стандартный номер протокола */
}
```

Пример:

Пусть необходимо найти официально назначенный номер порта протокола для SMTP.

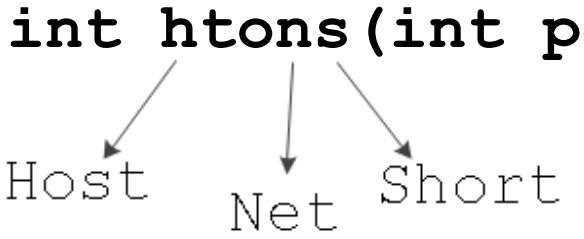
```
struct protoent *ppe;

 if  ( ppe = getprotobyname("udp"))
   {
        /* Теперь стандартный номер протокола  в    ppe–>p_proto */
    }
  else      /* Возникла ошибка - обработать ошибку */
```

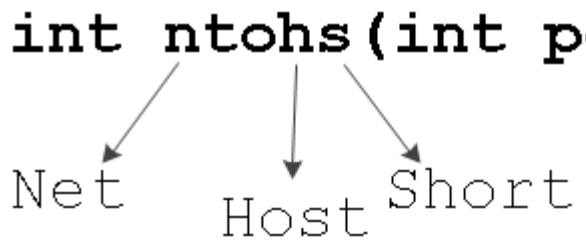
API Socket

(Преобразование числа в сетевой/прямой порядок байт)

Преобразование числа
из **прямого** порядка
в **сетевой** порядок байт

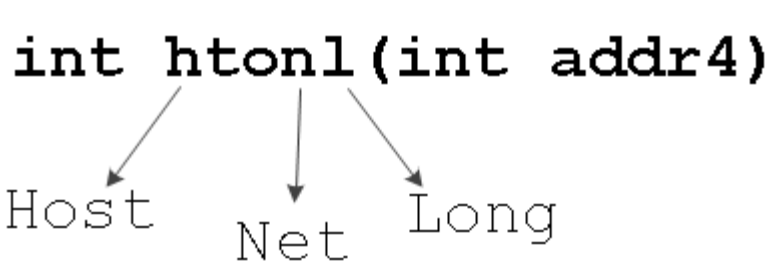


Преобразование числа
из **сетевого** порядка
в **прямой** порядок байт

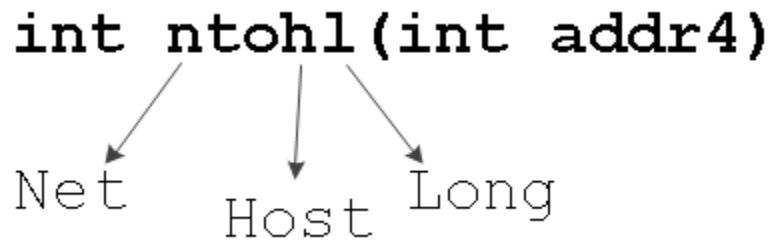


2 байта
0…65535

Преобразование числа
из **прямого** порядка
в **сетевой** порядок байт



Преобразование числа
из **сетевого** порядка
в **прямой** порядок байт



4 байта
$0\ldots 2^{32}-1$

Нахождение свободного порта
Функция getsockname

```
struct sockaddr_in    servAddr;

                    ……….

    servAddr.sin_family = AF_INET;
    servAddr.sin_addr.s_addr = htonl( INADDR_ANY );
    servAddr.sin_port = 0;


bind( sockMain, &servAddr,  sizeof(servAddr));

getsockname( sockMain,  &servAddr,   &length  );

printf( "СЕРВЕР: номер порта - % d\n",   ntohs(servAddr.sin_port )  );

                    ……..
```

Функции преобразования IP-адресов из строк ASCII в двоичные числа с сетевым порядком байтов (хранящиеся в структурах адресов сокетов) и обратно. `inet_aton`

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

int inet_aton(const char *strptr, struct in_addr *addrptr);
```

Возвращает: 1, если строка преобразована успешно, 0 в случае ошибки.

Функция преобразует строку, на которую указывает `strptr`, в 32-разрядное двоичное число (например, 172.103.15.189), записанное в сетевом порядке байтов, передаваемое через указатель `addrptr`. В случае успешного выполнения возвращаемое значение равно 1, иначе возвращается нуль.

Примечание. *Функция `inet_aton` обладает одним недокументированным свойством: если `addrptr` — пустой указатель (null pointer), функция все равно выполняет проверку допустимости адреса, содержащегося во входной строке, но не сохраняет результата.*

Функции преобразования IP-адресов. `inet_addr`

```
in_addr_t inet_addr(const char *strptr);
```

Возвращает: 32-разрядный адрес IPv4 в сетевом порядке байтов: INADDR_NONE в случае ошибки.

Функция выполняет такое же преобразование, что и `inet_aton`, возвращая в качестве значения 32-разрядное двоичное число в сетевом порядке байтов. Проблема при использовании этой функции состоит в том, что все 232 возможных двоичных значений являются действительными IP-адресами (от 0.0.0.0 до 255.255.255.255), но в случае возникновения ошибки функция возвращает константу INADDR_NONE.

Функция является нерекомендуемой, или устаревшей, и в создаваемом коде вместо нее **следует использовать функцию** `inet_aton`.

Функции преобразования IP-адресов. `inet_ntoa`

Функция `inet_ntoa` является обратной к вышеописанным.

```
char *inet_ntoa(struct in_addr inaddr);
```

возвращает: указатель на строку с адресом в точечно-десятичной записи.

Функция преобразует 32-разрядный двоичный адрес IPv4, хранящийся в сетевом порядке байтов, в точечно-десятичную строку. Строка, на которую указывает возвращаемый функцией указатель, находится в статической памяти. Это означает, что функция не допускает повторного вхождения, то есть не является повторно входимой (reentrant). Наконец, отметим, что *эта функция принимает в качестве аргумента структуру, а не указатель на структуру*.

Функции преобразования IP-адресов. inet_pton

Более новые функции inet_pton и inet_ntop работают как с адресами IPv4, так и с адресами IPv6..

```
int inet_pton(int af, const char *src, void *dst);
```

Данная функция преобразует строку символов src в сетевой адрес (типа af), затем копирует полученную структуру с адресом в dst.

На текущий момент **поддерживаются следующие типы адресов:**

AF_INET – указывает на строку символов, содержащую сетевой адрес IPv4 в формате "ddd.ddd.ddd.ddd". Адрес преобразуется в struct in_addr и копируется в переменную dst, размер которой должен быть равен sizeof(struct in_addr) байтам.

AF_INET6 – указывает на строку символов, содержащую адрес сети IPv6 в разрешенном для сети IPv6 формате адреса. Адрес преобразуется в struct in6_addr и копируется в переменную dst, размер которой должен быть sizeof(struct in6_addr) байтов.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

inet_pton возвращает отрицательное значение и меняет значение переменной errno на EAFNOSUPPORT, если af не содержит правильного типа адреса. Возвращается 0, если src не содержит строку символов, представляющую правильный сетевой адрес (для указанного типа адресов). Если сетевой адрес был успешно преобразован, то возвращается положительное значение.

Функции преобразования IP-адресов. `inet_ntop`

```
const char *inet_ntop(int af, const void *src, char *dst, size_t cnt);
```

Данная функция преобразует структуру сетевого адреса `src` в строку символов с сетевым адресом (типа `af`), которая затем копируется в символьный буфер `dst`; размер этого буфера составляет `cnt` байтов.

На текущий момент **поддерживаются следующие типы адресов:**

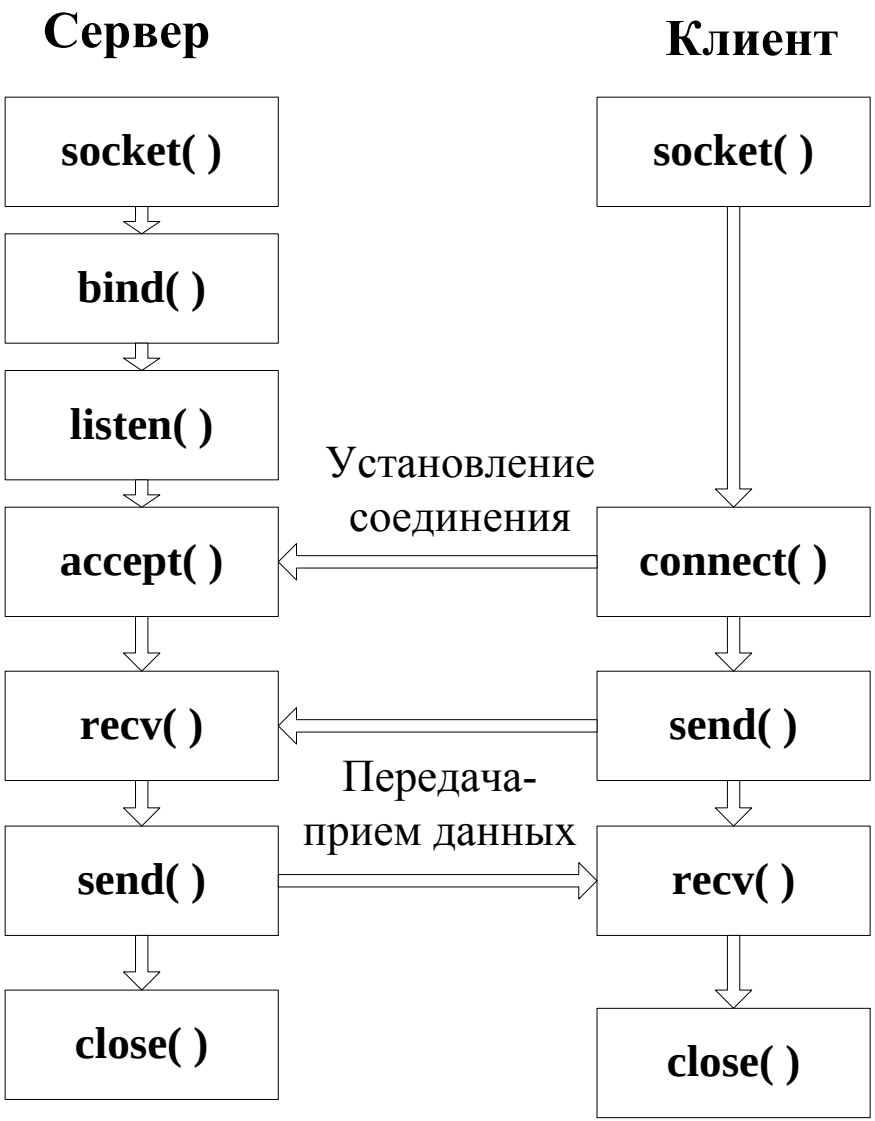
`AF_INET` (для `src`) указывает на `struct in_addr` (в формате сетевого порядка расположения байтов), которая преобразуется в IPv4-сетевой адрес в формате "ddd.ddd.ddd.ddd". Буфер `dst` должен быть размером, по меньшей мере, равным `INET_ADDRSTRLEN` байтам.

`AF_INET6` (для `src`) указывает на `struct in6_addr` (в формате сетевого порядка расположения байтов), которая преобразуется в представление этого адреса в наиболее верном формате IPv6. Буфер `dst` должен быть размером, по меньшей мере, равным `INET6_ADDRSTRLEN` байтам.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

`inet_ntop` возвращает ненулевой указатель на `dst`. В случае ошибок возвращается NULL, а также `errno` присваивается значение `EAFNOSUPPORT`, если значение `af` не было установлено равным корректному типу адреса или равным `ENOSPC`, если полученная после преобразования строка превышает размер `dst` (заданный параметром `cnt`).

Программа типа клиент-сервер для TCP



Программа сервера TCP (начало)

```
#include <sys/types.h>        #include <sys/socket.h>        #include <stdio.h>
#Include <netinet/in.h>       #include <netdb.h>             #include <errno.h>

main() {
  int  sockMain,   sockClient,    length, child;
  struct sockaddr_in    servAddr;

   if(( sockMain = socket( AF_INET,  SOCK_STREAM, 0 ) )  < 0 )   {
                perror("Сервер не может открыть главный socket."); exit(1);
        }
         bzero( (char *) &servAddr, sizeof( servAddr ) );
      servAddr.sin_family = AF_INET;
      servAddr.sin_addr.s_addr = htonl( INADDR_ANY );
      servAddr.sin_port = 0;
    if ( bind( sockMain, &servAddr,  sizeof(servAddr) ) )  {
                perror("Связывание сервера неудачно.");   exit(1);
        }
      length = sizeof( servAddr ) ;
    if ( getsockname( sockMain,  &servAddr,   &length  ) )  {
                perror("Вызов getsockname неудачен."); exit(1);
        }
    printf( "СЕРВЕР: номер порта - % d\n",   ntohs(servAddr.sin_port )  )
    listen(sockMain, 5) ;
```

Программа сервера TCP (продолжение)

```
for(  ; ;  )  {
        if ( ( sockClient   =   accept( sockMain,  0,  0 ) )  <  0 ) {
                perror("Неверный socket для      клиента."); exit(1);
           }
          BuffWork(sockClient);
             close(sockClient) ;
    }
 }

#define BUFLEN 81

int      BuffWork(int sockClient)
{
    char buf[BUFLEN];
    int msgLength;
         bzero(buf, BUFLEN);
        if( ( msgLength = recv( sockClient, buf, BUFLEN, 0 ) ) < 0 ){
                perror("Плохое  получение дочерним процессом."); exit(1);
            }
        printf( "SERVER: Socket для клиента - %d\n",   sockClient);
        printf( "SERVER: Длина сообщения - %d\n",   msgLength);
        printf( "SERVER: Сообщение:  %s\n\n",   buf);
 }
```

Программа клиента TCP

```
main( argc, argv )
int argc;  char* argv [ ] ;  {
  int sock;
  struct sockaddr_in servAddr;
  struct hostent *hp, *gethostbyname();
          if(argc < 4)  {   printf("ВВЕСТИ tcpclient имя_хоста порт сообщение\n"); exit(1) ;
                  }
             if((sock = socket( AF_INET,  SOCK_STREAM, 0  ) ) < 0) {
                        perror("Не могу получить       socket\n"); exit(1);
                    }
             bzero((char *)&servAddr,sizeof(servAddr)) ;
            servAddr.sin_family  =  AF_I.NET ;
                 hp = gethostbyname( argv [1] ) ;
            bcopy( hp -> h_addr,  &servAddr.sin_addr,  hp -> h_length ) ;
            servAddr.sin_port = htons( atoi(argv[2] ) ) ;
          if( connect( sock, &servAddr,sizeof( servAddr ) ) < 0 ){
                        perror("Клиент не может соединиться.\n"); exit(1);
                   }
            printf  ("CLIENT: Готов к пересылке\n") ;
         if( send ( sock, argv[3], strlen(argv[3]), 0 ) < 0 ) {
                        perror( "Проблемы с пересылкой.\n" ); exit(1);
                  }
           printf("CLIENT: Пересылка завершена. Счастливо оставаться.\n");
        close(sock);    exit(0);   }
```

Функция fork()

https://www.opennet.ru/docs/RUS/linux_parallel/node7.html
https://ru.wikipedia.org/wiki/Fork

Системный вызов `fork()` создаёт новый процесс. Синтаксис вызова следующий:

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

pid_t fork(void);
```

`pid_t` является примитивным типом данных, который **определяет идентификатор процесса** или группы процессов. При вызове `fork()` порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю.

Процесс-потомок наследует следующие признаки родителя:

- сегменты кода, данных и стека программы;
- таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения;
- рабочий и корневой каталоги;
- реальный и эффективный номер пользователя и номер группы;
- приоритеты процесса (администратор может изменить их через `nice`);
- контрольный терминал;
- маску сигналов;
- ограничения по ресурсам;
- сведения о среде выполнения;
- разделяемые сегменты памяти.

Потомок не наследует от родителя следующих признаков:

- идентификатора процесса (PID, PPID);
- израсходованного времени ЦП (оно обнуляется);
- сигналов процесса-родителя, требующих ответа;
- блокированных файлов (record locking).

Функция fork()

При успешном вызове `fork()` в системе выполняются, два почти «идентичных» процесса.
Весь код после `fork()` выполняется, как в процессе-потомке, так и в процессе-родителе.

```
PID = fork();
```

`If PID < 0;` **/* Ошибка при порождении процесса. */**

Процесс-потомок и процесс-родитель получают разные коды возврата после вызова `fork()`.

`If PID > 0;` **/* Это процесс-родитель**, который успешно создал процесс-потомок. Значение записанное в `PID` – идентификатор созданного процесса-потомка. ***/**

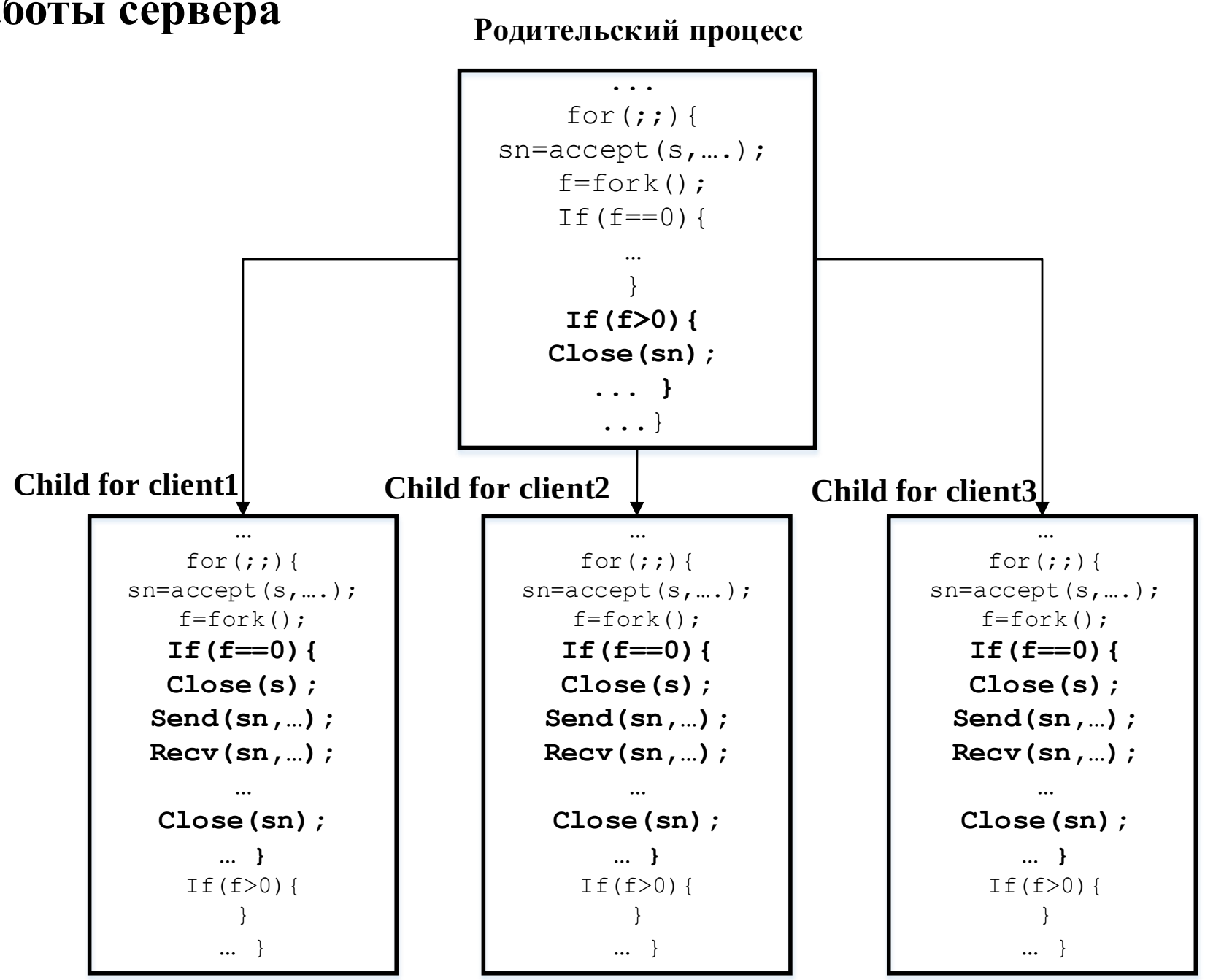
`If PID == 0;` **/* Это процесс-потомок.**
Процесс-потомок получает в качестве кода возврата значение 0, если вызов `fork()` оказался успешным. ***/**

Применение процессов для обеспечения параллельной работы сервера

```
/* Создаем ведущий сокет, привязываем его к общепринятому порту и переводим в пассивный режим */

for( ; ; ) {
    if( sockClient  =  accept( sockMain, (struct sockaddr *)&fsin, &len) )  <  0 )
        {   /* Сообщение об ошибке */
            exit(1);
         }
    child  =  fork()  ;
     if( child  <  0 )
        {    /* Сообщение об ошибке */
            exit(1);
        }
     if(  child  = =  0 )
       {      close( sockMain ) ;
                /* Обработка поступившего запроса процессом */
              close( sockClient ) ;
              exit( 0 ) ;
        }
    close( sockClient ) ;
  }
```

Применение процессов для обеспечения параллельной работы сервера



Применение процессов для обеспечения параллельной работы сервера

Пример 2

```
signal( SIGCHLD, reaper ) ;
while(1) {
        ssock = accept( msock, (struct sockaddr *)&fsin,  &len) ;
          if( ssock < 0 ){   /* Сообщение об ошибке */ }
              switch( fork() ) {
                    case( 0 ) :
                              close( msock ) ;
                               /* Обработка поступившго запроса ведомым процессом */
                              close( ssock ) ;
                               exit( 0 ) ;
                     default:
                               close( ssock ) ;
                                /* Ведущий процесс */
                                break;
                      case ( -1 ) :
                                /* Сообщение об ошибке */
                    }
               }

  void reaper( int sig )
   {   int status;
              while( wait3( &status, WNOHANG, (struct rusage *) 0 ) >=  0 ) ;
   }
```

Применение процессов для обеспечения параллельной работы сервера

Замечание: первоначальный, и новые процессы имеют доступ к открытым сокетам после вызова функции **fork()** и они оба должны закрыть эти сокеты, после чего система освобождает связанные с ним ресурсы.

Завершившийся процесс остается в виде так называемого процесса-зомби до тех пор, пока родительским процессом не будет выполнен системный вызов **wait3**. Для полного завершения дочернего процесса (т.е. уничтожения процесса-зомби) необходимо перехватывается сигнал завершения дочернего процесса. Операционной системе дается указание, что для ведущего процесса сервера при получении каждого сигнала о завершении работы дочернего процесса (сигнал **SIGCHLD**) должна быть выполнена функция reaper, в виде следующего вызова: **signal(SIGCHLD, reaper);**

Функция reaper вызывает системную функцию **wait3**, которая остается заблокированной до тех пор, пока не произойдет завершение работы одного или нескольких дочерних процессов. Эта функция возвращает значение структуры **status**, которую можно проанализировать для получения дополнительной информации о завершившемся процессе. Поскольку данная программа вызывает функцию **wait3** при получении сигнала **SIGCHLD**, вызов этой функции всегда происходит только после завершения работы дочернего процесса. Для предотвращения возникновения в сервере тупиковой ситуации в случае ошибочного вызова в программе используется параметр **WNOHANG**, который указывает, что функция **wait3** не должна блокироваться в ожидании завершения какого-либо процесса. ***Можно узнать о наличии зомби-процесса командой:*** `>ps -aux`

Если в обработчике *SIGCHLD* нет ничего кроме вызова *wait*, то можно просто установить этот обработчик в ***SIG_IGN***. Тут следует заметить, что установка *SIGCHLD* в *SIG_IGN* совместима с POSIX.1-2001, и не совместима с POSIX.1-1990.

СПАСИБО ЗА ВНИМАНИЕ

Применение потоков для обеспечения параллельной работы сервера

- **int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);**

  После создания нового потока в нем начинается выполняться функция (которая называется потоковой функцией), переданная параметром **start_routine**, причем ей самой в качестве первого параметра передается переменная **arg**. Параметр attr позволяет задать атрибуты потока (**NULL** для значений по умолчанию). **thread** -- адрес переменной, в которую **pthread_create()** записывает идентификатор созданного потока. Созданный с помощью pthread_create() поток будет работать параллельно с существующими. Возвращается 0 в случае успеха и не ноль -- в противоположном случае. Потоковая функция **start_routine** имеет прототип:

- **void* my_thread_function(void *);**

  Поскольку как параметр, так и ее возвращаемое значение -- указатели, то функция может принимать в качестве параметра и возвращать любую информацию.

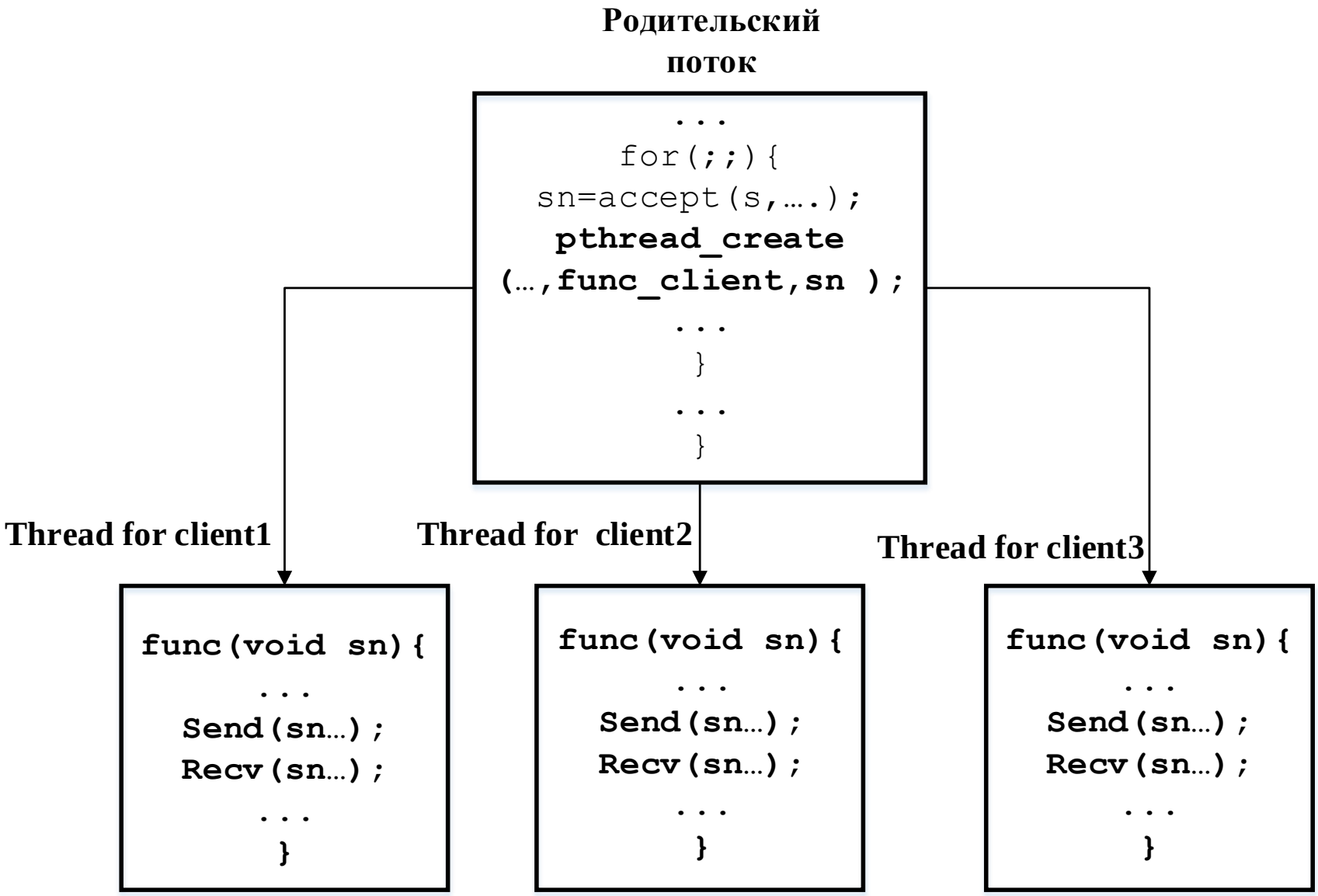
Применение потоков для обеспечения параллельной работы сервера

- Выполнение потока завершается в двух случаях: если завершено выполнение потоковой функции, или при выполнении функции **pthread_exit()**:

- **void pthread_exit(void *retval);**

которая завершает выполнение вызвавшего ее потока. Аргумент **retval** -- это код с которым завершается выполнение потока. При завершении работы потока вы должны помнить, что **pthread_exit()** не закрывает файлы и все открытые потоком файлы будут оставаться открытыми даже после его завершения, так что не забывайте подчищать за собой. Если вы завершите выполнение функции **main()** с помощью **pthread_exit()**, выполнение порожденных ранее потоков продолжится.

Применение потоков для обеспечения параллельной работы сервера



Применение потоков для обеспечения параллельной работы сервера

- Потоки, как и порожденные процессы, по завершению работы сами по себе не освобождают ресурсы, занятые собой для личного пользования (а именно дескриптор и стек) :-) . Поэтому им необходимо помочь.

  Варианта, собственно, два: либо на ряду с освобождением ресурсов какой-либо поток ждет его завершения, либо нет.

  Для первого варианта используем функцию **pthread_join():**

- **int pthread_join(pthread_t th, void **thread_return);**

которая приостанавливает выполнение вызвавшего ее процесса до тех пор, пока поток, определенный параметром th, не завершит свое выполнение и если параметр thread_return не будет равен NULL, то запишет туда возвращенное потоком значение (которое будет равным либо **PTHREAD_CANCELED**, если поток был отменен, либо тем значением, которое было передано через аргумент функции **pthread_exit()**).

Применение потоков для обеспечения параллельной работы сервера

- Для второго варианта есть функция **pthread_detach() :**

- **int pthread_detach(pthread_t th);**

  которая делает поток **th** "открепленным" (detached). Это значит, что после того, как он завершится, он сам освободит все занятые ним ресурсы.

  Обратите внимание на то, что нельзя ожидать завершения detached-потока (то есть функция **pthread_join** выполненная для detached потока завершится с ошибкой).

Применение потоков для обеспечения параллельной работы сервера

- Создание потоков позволяет им совместно использовать некоторые ресурсы, но нужно производить контроль на предмет эксклюзивности доступа к этим ресурсам (нельзя допускать одновременной записи в одну переменную, например).

  Такой контроль можно вести тремя способами через:

- ***взаимоисключающую блокировку***
- ***условные переменные***
- ***Семафоры***

Применение потоков для обеспечения параллельной работы сервера

- Первый вариант представляется самим набором функций библиотеки **POSIX Threads**. Взаимоисключающая блокировка представляется в программе переменной типа **pthread_mutex_t**.

  Для работы с ними существует 5 функций, а именно:

- **int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);**

  которая инициализирует блокировку, заданную параметром **mutex.** Соответственно, **mutexattr** -- ее атрибуты. Значение **NULL** соответствует установкам по умолчанию.

- **int pthread_mutex_destroy(pthread_mutex_t *mutex);**

  удаляет блокировку mutex

Применение потоков для обеспечения параллельной работы сервера

- **int pthread_mutex_lock(pthread_mutex_t *mutex)**

  устанавливает блокировку **mutex**. Если **mutex** не была заблокирована, то она его ее и немедленно завершается. Если же нет, то функция приостанавливает работы вызвавшего ее потока до разблокировки mutex, а после этого выполняет аналогичные действия.

- **int pthread_mutex_unlock(pthread_mutex_t *mutex)**

  снимает блокировку **mutex**. Подразумевается, что эта функция будет вызвана тем же потоком, который ее заблокировал (через **pthread_mutex_lock()**).

- **int pthread_mutex_trylock(pthread_mutex_t *mutex)**

  ведет себя аналогично **pthread_mutex_lock()** за исключением того, что она не приостанавливает вызывающий поток, если блокировка **mutex** установлена, а просто завершается с кодом **EBUSY**.

Применение потоков для обеспечения параллельной работы сервера

- Поток выполнения представляет собой один из принципов организации отдельных вычислений, а один процесс может содержать от одного и более потоков.

- Новый поток может быть создан в любое время путем вызова функции **pthread_create**.

- Операционная система ограничивает максимально допустимое количество параллельных потоков, также как и максимальное количество параллельных процессов.

- Все потоки процесса разделяют единый набор глобальных переменных и единый набор дескрипторов файлов.

- Многопотоковые процессы обладают двумя основными **преимущества**ми по сравнению с однопотоковыми процессами:

- ***более высокая эффективность и разделяемая память.***

Повышение эффективности связано с уменьшением издержек на переключение контекста.

Применение потоков для обеспечения параллельной работы сервера

- *Переключение контекста –*

это действия, выполняемые операционной системой при передаче ресурсов процессора от одного потока выполнения к другому.

При переключении с одного потока на другой операционная система должна сохранить в памяти состояние предыдущего потока (например, значения регистров) и восстановить состояние следующего потока.

Потоки в одном и том же процессе разделяют значительную часть информации о состоянии процесса, поэтому операционной системе приходится выполнять меньший объем работы по сохранению и восстановлению состояния.

Вследствие этого переключение с одного потока на другой в одном и том же процессе происходит быстрее по сравнению с переключением между двумя потоками в разных процессах.

Применение потоков для обеспечения параллельной работы сервера

- Второе преимущество потоков (***разделяемая память***), является более важным, чем повышение эффективности.

Потоки упрощают разработку параллельных серверов, в которых все копии сервера должны взаимодействовать друг с другом или обращаться к разделяемым элементам данных. В частности, поскольку ведомые потоки в сервере совместно используют глобальную память.

Одним ***из недостатков потоков является*** то, что они имеют ***общее состояние процесса***, поэтому действия, выполненные одним потоком, могут повлиять на другие потоки в том же процессе. Например, если два потока попытаются одновременно обратиться к одной и той же переменной, они могут помешать друг другу. API-интерфейс потоков предоставляет функции, которые могут использоваться потоками для координации работы.

Применение потоков для обеспечения параллельной работы сервера

- Еще один ***недостаток*** связан с ***отсутствие***м ***надежности***.

Если одна из параллельно работающих копий однопотокового сервера вызовет серьезную ошибку (например, в ней будет выполнена ссылка на недопустимую область памяти), то операционная система завершит только тот процесс, который вызвал ошибку. С другой стороны, если серьезная ошибка будет вызвана одним из потоков многопотокового сервера, то операционная система завершит весь процесс (т.е. все потоки этого процесса).

Пример сервера, реализованного с применением потоков

```
pthread_mutex_t st_mutex;   /* Разделяемая переменная */

int main() {
pthread_t    th;
pthread_attr_t    ta;
        /* Создаем ведущий сокет, привязываем его к общепринятому порту и
            переводим   в пассивный режим */
pthread_attr_init(&ta);
pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
pthread_mutex_init(&st_mutex,0);
while (1)  {ssock = accept(msock,  (struct sockaddr *)&fsin, &len);
      if (ssock <0)  {/* ошибка */}
      if (pthread_create( &th,  &ta, (void *)handler,  (void *) ssock)  < 0 )
        {          /* ошибка */}
         }


 int handler ( int ssock )
{ pthread_mutex_lock(&st_mutex);
/* выполнение операций с разделяемыми переменными */
     pthread_mutex_unlock(&st_mutex);
     return 0;
}
```

Применение потоков для обеспечения параллельной работы сервера

- Аргумент **attr** содержит атрибуты, присваиваемые вновь создаваемому потоку. Значение аргумента может быть равно **NULL**, если новый поток должен использовать атрибуты, принятые системой по умолчанию, или адрес объекта содержит атрибуты. Объект, содержащий атрибуты, может быть связан с несколькими потоками.
- Функция **pthread_attr_init** создает объект, содержащий атрибуты, а функция **phread_attr_destroy** удаляет такой объект:

  #include <pthread.h>

  Int pthread_attr_init(pthread_attr_t* attr_p**);**

  Int pthread_attr_destroy(pthread_attr_t* attr_p**);**

- Атрибуты объекта, созданного функцией **pthread_attr_init**, можно проверить функцией **pthread_attr_get**, или установить функцией **pthread_attr_set** .
- Например, состояние отсоединения

  API для проверки - **pthread_attr_getdetachstate**

  API для установки - **pthread_attr_setdetachstate**

Применение потоков для обеспечения параллельной работы сервера

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define NTHRDS 2  /* К этим ресурсам мы можем получить   доступ из потока */

char words[8000][20]; // Словарь
int current, maxw;
char foundpass; // Флажок

pthread_mutex_t mutexpass, mutexfound; /* Блокировки */

void *passhack(void *hash)

int main(int argc, char **argv)
{
char i, *pass;
FILE *passfile;
```

Применение потоков для обеспечения параллельной работы сервера

```
pthread_t thrds[NTHRDS];   // Потоки
pass = (char *)malloc(35);
pthread_mutex_init(&mutexpass, NULL); /* Инициализация блокировок */
pthread_mutex_init(&mutexfound, NULL);
     strcpy(pass, argv[1]);  /* Читаем словарь в память */
     strcpy(pass, argv[1]);  /* Читаем словарь в память */
     passfile = fopen(argv[2], "r");
     maxw = 0;
    while ( !feof(passfile))  fgets(words[maxw++], 20, passfile);
                    fclose(passfile);
    foundpass = 0;
/* Запускаем потоки */
for ( i=0 ;   i < NTHRDS;   i++ )
          pthread_create(&thrds[i], NULL, passhack, (void *) pass);
                        /* И ждем завершения их работы */
    for ( i=0 ; i < NTHRDS;   i++ )
              pthread_join(thrds[i], NULL);

/* Освобождаем блокировки */
   pthread_mutex_destroy(&mutexpass);
   pthread_mutex_destroy(&mutexfound);
         return 0;
}
```

Применение потоков для обеспечения параллельной работы сервера

- Аргумент **attr** содержит атрибуты, присваиваемые вновь создаваемому потоку. Значение аргумента может быть равно **NULL**, если новый поток должен использовать атрибуты, принятые системой по умолчанию, или адрес объекта содержит атрибуты. Объект, содержащий атрибуты, может быть связан с несколькими потоками.
- Функция **pthread_attr_init** создает объект, содержащий атрибуты, а функция **phread_attr_destroy** удаляет такой объект:

```
#include <pthread.h>
Int pthread_attr_init(pthread_attr_t*  attr_p);
Int pthread_attr_destroy(pthread_attr_t*  attr_p);
```

Применение потоков для обеспечения параллельной работы сервера

- Атрибуты объекта, созданного функцией **pthread_attr_init**, можно проверить функцией **pthread_attr_get**, или установить функцией **pthread_attr_set** .
- **Например,**

состояние отсоединения

API для проверки - **pthread_attr_getdetachstate**

API для установки - **pthread_attr_setdetachstate**

правила планирования

API для проверки - **pthread_attr_getschedpolicy**

API для установки – **pthread_attr_setschedpolicy**

Правила планирования задают, среди прочего, приоритет потока

Параметры планирования

API для проверки - **pthread_attr_getschedparam**

API для установки - **pthread_attr_setschedparam**

Второй аргумент в **pthread_attr_getschedparam** и **pthread_attr_setschedparam** *– это адрес переменной типа* ***struct sched_param*** *. В этой переменной есть целочисленное поле* ***sched_priority****, в котором задается приоритет любогопотока, обладающего этим свойством.*