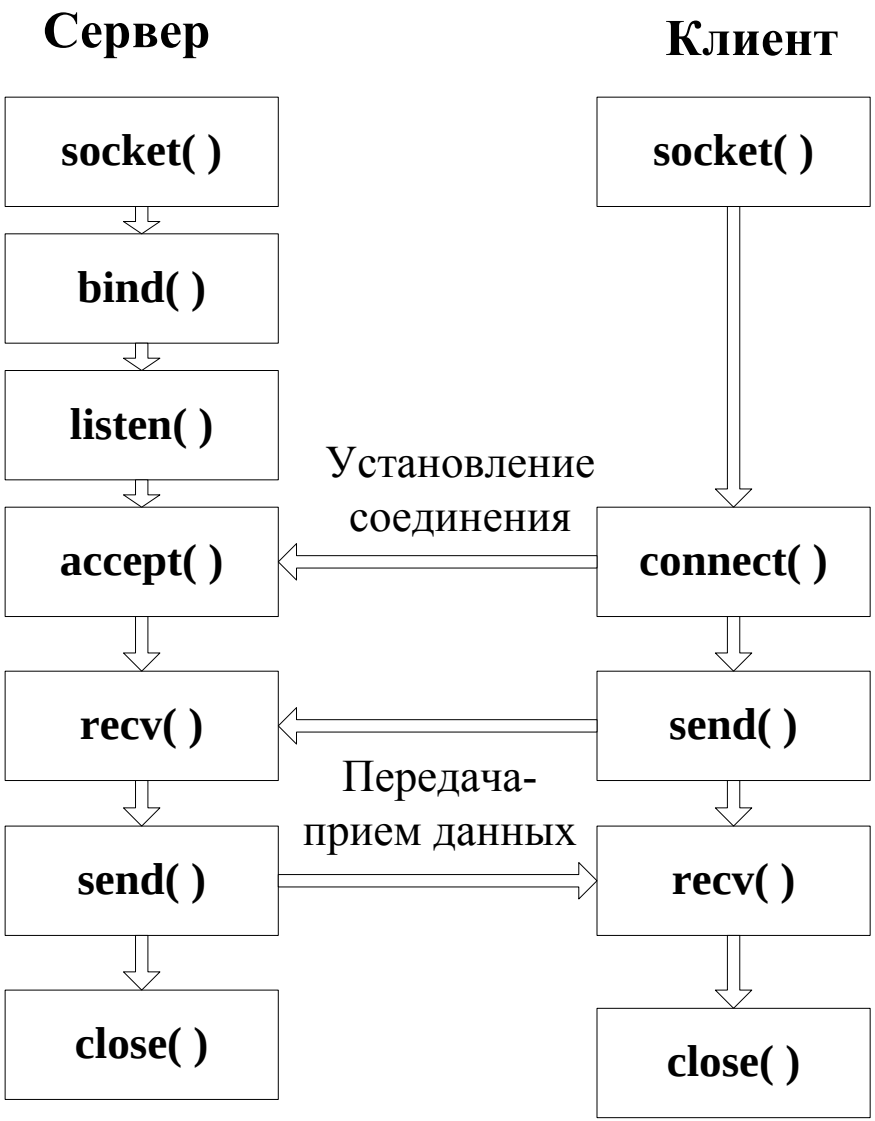
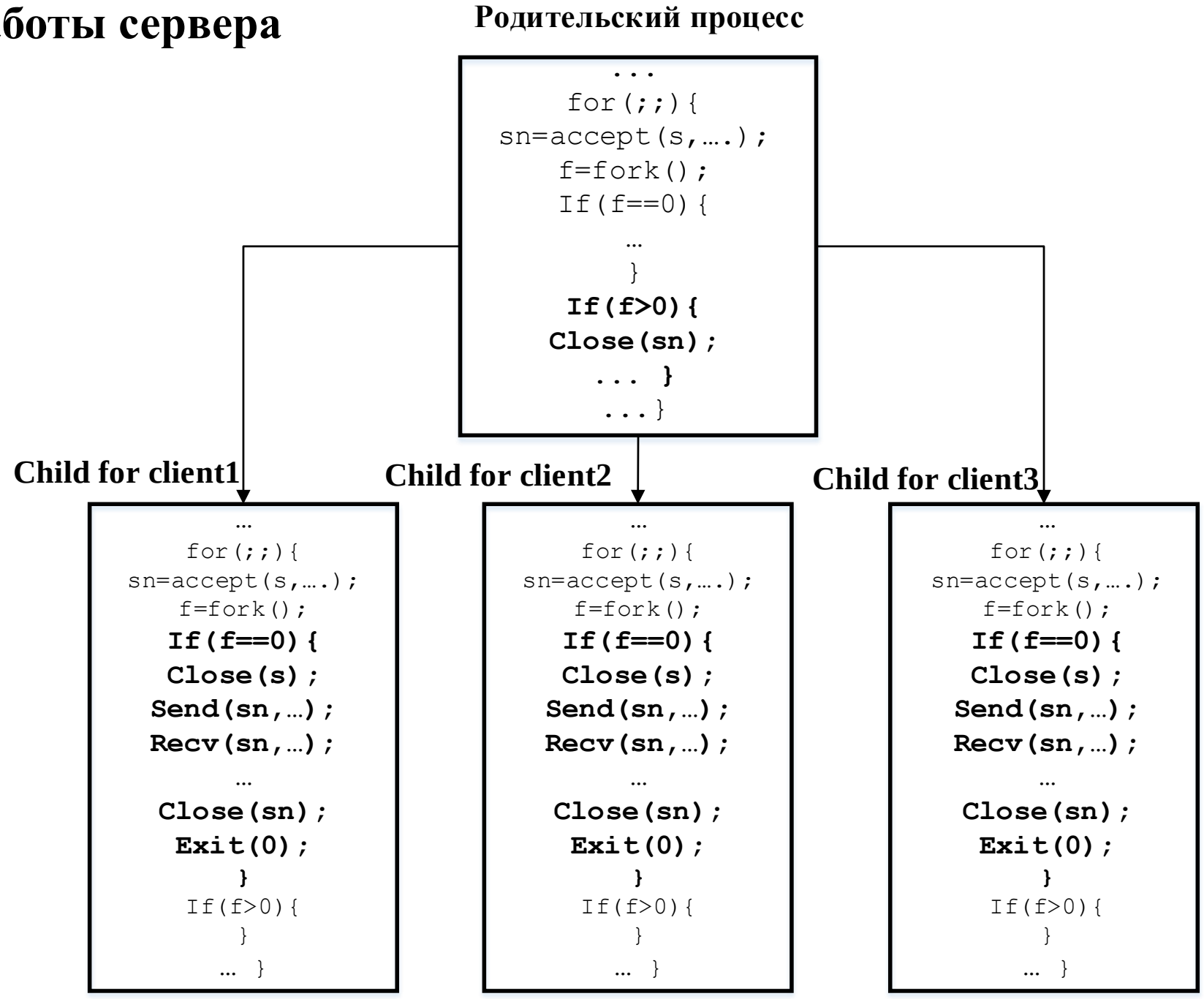
Программа типа клиент-сервер для TCP



Применение процессов для обеспечения параллельной работы сервера



Применение процессов для обеспечения параллельной работы сервера

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

Родительский процесс

```
...
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
…
}
If(f>0){
Close(sn);
... }
...}
```

Можно узнать о наличии зомби-процесса командой: `>ps –aux`

Child for client1

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
Exit(0);
}
If(f>0){
}
… }
```

Child for client2

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
Exit(0);
}
If(f>0){
}
… }
```

Child for client3

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
Exit(0);
}
If(f>0){
}
… }
```

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.

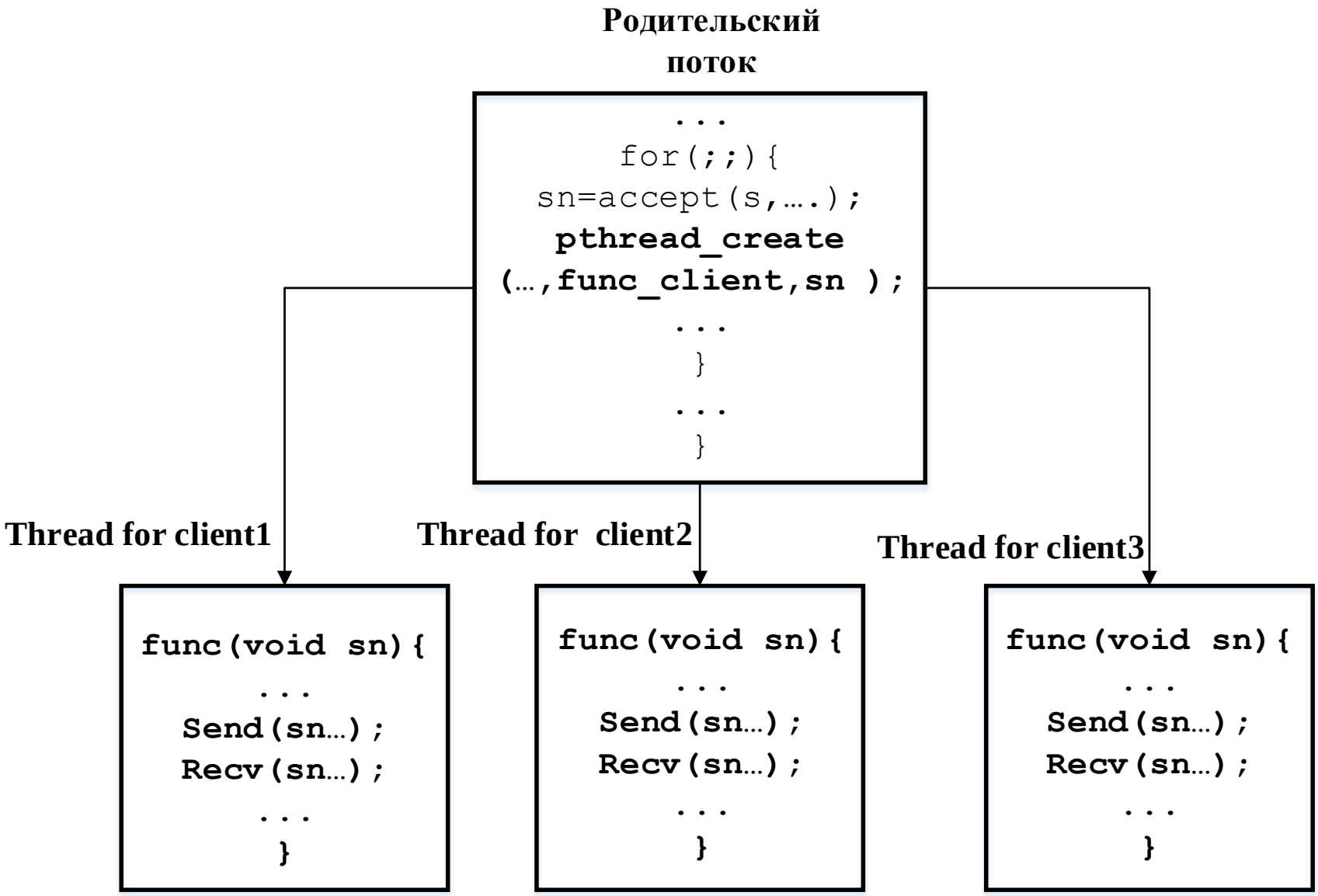
Применение потоков для обеспечения параллельной работы сервера

- 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, handler(void *), (void *) ssock)  < 0 )
          {          /* ошибка */},
           }


 int handler ( int ssock )
{  recv(ssock, ….);
    pthread_mutex_lock(&st_mutex);
    /* выполнение операций с разделяемыми переменными */ fwrite (…);
        pthread_mutex_unlock(&st_mutex);
        …..;  close(ssock);    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

- 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

```
pthread_mutex_t mut;

int main() {
pthread_t thr_id;
int  Mainsock;
int Clientsock;

/* Инициализация пассивного сокета Mainsock*/
…
pthread_mutex_init(&mut,0);

while (1)  {
Clientsock = accept(Mainsock, (struct sockaddr *)&fsin, &len);
if (Clientsock <0) {/* Обработка ошибки accept*/}
if (pthread_create(&thr_id, NULL, (void *) threadclient(void *), (void *) Clientsock) <0) {/* Обработка ошибки pthread_create */}
}
}//end main()

int threadclient (int sock)   {
  pthread_mutex_lock(&mut);
/* Выполнение операций с разделяемым пространством*/
   pthread_mutex_unlock(&mut); ….; close(sock);
return 0;
}
```

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

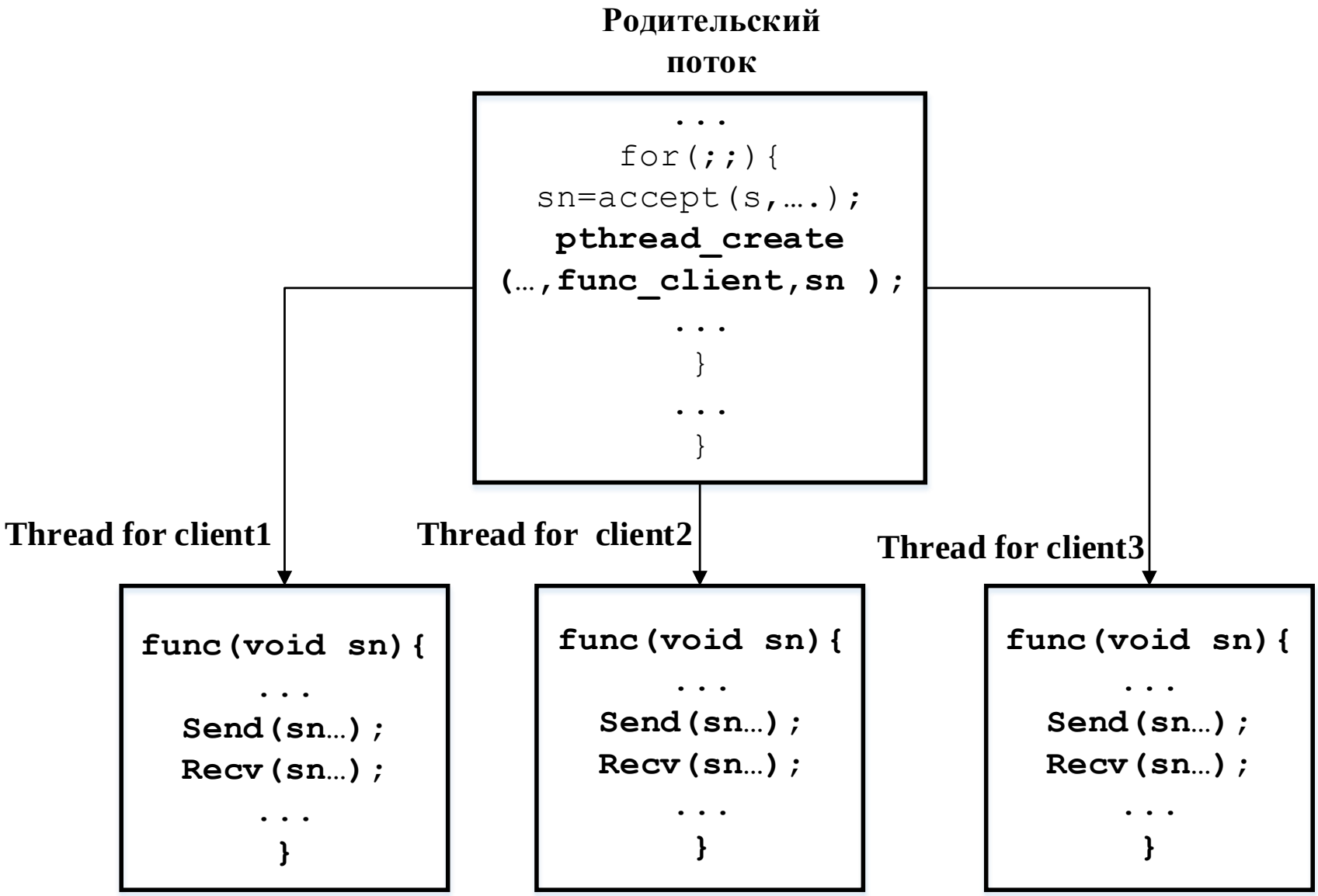
Параметры планирования

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.*

Применение потоков для обеспечения параллельной работы сервера



Однопотоковые псевдопараллельные серверы Особенности

- Необходимо предусмотреть, чтобы единственный поток выполнения в сервере держал открытыми соединения с несколькими клиентами и обеспечивал обслуживание сервером того соединения, через которое в определенный момент поступают данные.

- Однопотоковая реализация не требует переключения между контекстами потоков или процессов, поэтому она может дан выдержать более высокую нагрузку по сравнению с реализацией, в которой используются несколько потоков или процессов.

Асинхронный ввод/вывод, организованный с помощь системного вызова select

- В основе разработки программы однопотокового, параллельного сервера лежит использование асинхронного ввода/вывода, организованного с помощь системного вызова **select**.

- Сервер создает сокет для каждого соединения, которое он должен поддерживать, а затем вызывает функцию **select**, которая ожидает поступления данных через каждое из них.

- Функция **select** может ожидать поступления запросов на выполнение операций ввода/вывода через все возможные сокеты, в том числе и одновременно ожидать поступления новых запросов на установление соединения.

Асинхронный ввод/вывод, организованный с помощь системного вызова select

- **НАЗВАНИЕ**

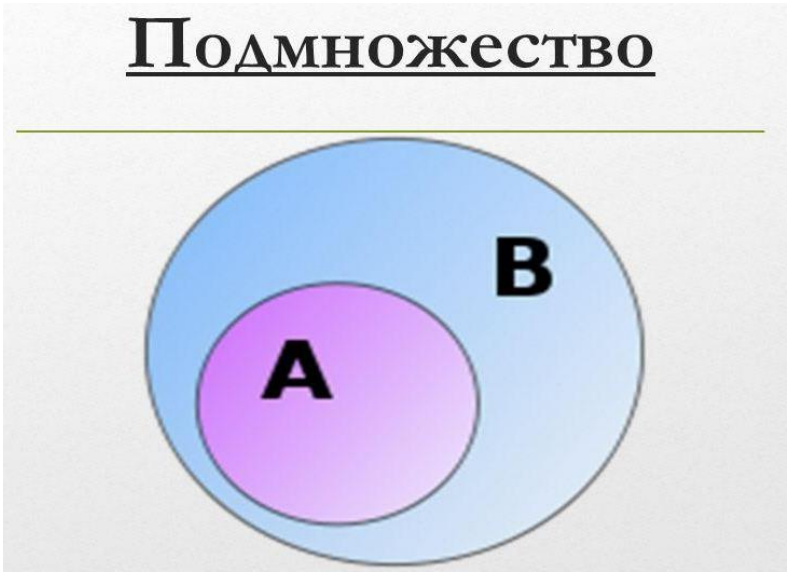
  select, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - синхронное мультиплексирование ввода-вывода

- **КРАТКАЯ СВОДКА**

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

- **int select(int** *n*, **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*);



Асинхронный ввод/вывод, организованный с помощь системного вызова select

ОПИСАНИЕ

- **select** ждет изменения статуса нескольких файловых дескрипторов.

  Отслеживаются три независимых набора дескрипторов. Те, что перечислены в параметре *readfds*, будут отслеживаться на предмет появления новых символов, доступных для чтения (говоря точнее, операция чтения не будет блокирована -- в частности, файловый дескриптор находится в конце файла); те, что указаны в параметре *writefds*, будут отслеживаться на предмет того, что операция записи не будет заблокирована; те же, что указаны в параметре *exceptfds*, будут отслеживаться на предмет исключительных ситуаций. При возврате из функции наборы дескрипторов модифицируются, чтобы показать, какие из них изменили свой статус.
- Для манипуляций наборами существуют четыре макроса: **FD_ZERO** очищает набор. **FD_SET** и **FD_CLR** добавляют или удаляют заданный дескриптор из набора. **FD_ISSET** проверяет, является ли дескриптор частью набора; этот макрос полезен после возврата из функции **select.**
- ***n*** на единицу больше самого большого номера дескриптора из всех наборов.
- ***timeout*** -- это верхняя граница времени, которое пройдет перед возвратом из **select**. Можно использовать ноль, при этом **select** завершится немедленно. Если **timeout** равен **NULL** (нет таймаута), то **select** будет ожидать изменений неопределенное время.

Пример. Системный вызов select

ПРИМЕР

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

 int  main ( void )
{  fd_set rfds;
   struct timeval   tv;
   int retval;
        /* Ждем, пока на стандартном вводе (fd 0) что-нибудь появится. */
   FD_ZERO(&rfds);
   FD_SET(0, &rfds);
         /* Ждем не больше пяти секунд. */
      tv.tv_sec = 5;
      tv.tv_usec = 0;
      retval = select(1, &rfds, NULL, NULL, &tv);       /* Не полагаемся на значение tv! */
   if (retval)
        printf("Данные доступны.\n");  /* Теперь FD_ISSET(0, &rfds) вернет истинное значение. */
   else
        printf("Данные не появились в течение пяти секунд.\n");
   exit(0);
}
```

Однопотоковые псевдопараллельные серверы

```
int msock;  /* Ведущий сокет сервера */
fd_set    rfds; /* Набор дескрипторов, готовых к чтению */
fd_set    afds; /* Набор активных дескрипторов */
int    fd,  nfds,   ssock  ;
                /*    инициализация пассивного сокета   msock     */
     nfds = getdtablesize();
 FD_ZERO(&afds);    FD_SET(msock,  &afds);
while (1) {
     memcpy(&rfds, &afds, sizeof(rfds));
      if (select(nfds, &rfds,  (fd_set *)0,  (fd_set *)0, (struct timeval *)0) <0) { /* ошибка */}
                /*       Блок 1      подключение клиента          */
          if ( FD_ISSET(msock, &rfds)  ) {       alen = sizeof(fsin);
               ssock = accept(msock,  (struct sockaddr *)&fsin, &alen); if (ssock <0) {/* ошибка */}
                  FD_SET(ssock, &afds);
               }
             /*       Блок 2     обработка запросов клиентов   */
        for ( fd = 0 ;  fd < nfds ;  fd++ )
            if ( fd != msock && FD_ISSET(fd, &rfds)  )
                 if ( handler(fd) == 0 ) { /* число полученных байт */
                      close(fd);
                        FD_CLR(fd,  &afds);
                    }
}
```

Однопотоковые псевдопараллельные серверы

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

- При успешном завершении **select** возвращает количество дескрипторов, находящихся в наборах дескрипторов, причем это количество может быть равным нулю, если таймаут истекает, а интересующие нас события так и не произошли. При ошибке возвращается -1, а *errno* устанавливается должным образом; наборы дескрипторов и значение *timeout* становятся неопределены, поэтому при ошибке нельзя полагаться на их значение.

ОШИБКИ

EBADF В одном из наборов находится неверный файловый дескриптор.

EINTR Был пойман незаблокированный сигнал.

EINVAL *n* отрицательно.

ENOMEM Функция **select** не смогла выделить участок памяти для внутренних таблиц.

ЗАМЕЧАНИЕ

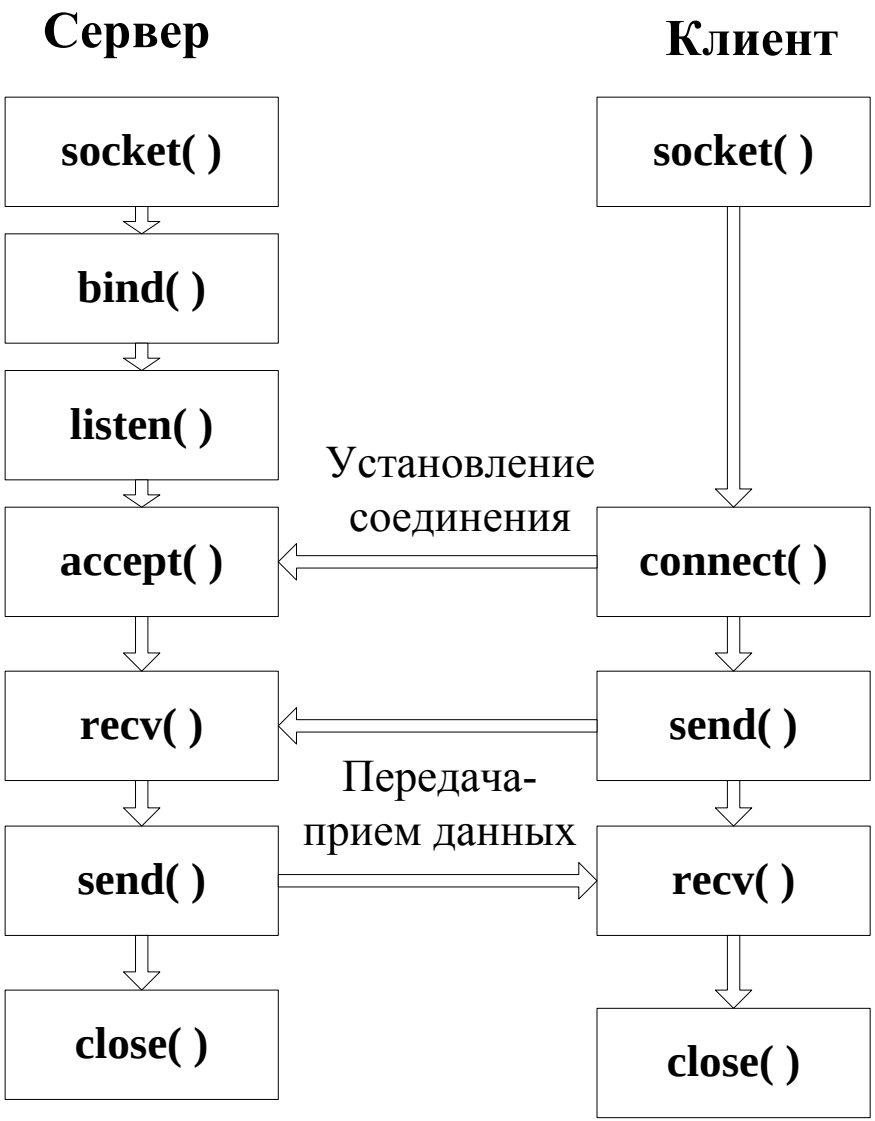
- В некоторых программах **select** вызывается с тремя пустыми наборами файлов, *n* равным нулю, и ненулевым значением *timeout*, что является довольно переносимым способом сделать задержку с миллисекундной точностью.
- Под Linux *timeout* изменяется, чтобы сообщить количество времени, которое не было использовано; большинство других реализаций не делают этого. Это приводит к проблемам как в коде под Linux, который читает значение *timeout* и переносится в другие операционные системы, так и когда код переносится под Linux и использует при этом struct timeval для нескольких функций **select** в цикле без повторной инициализации. Считайте, что параметр *timeout* неопределен после возврата из функции **select.**

Мультипротокольный сервер (TCP, UDP)

```
int    tsock; /* Ведущий сокет TCP */
int    usock; /* Сокет UDP */
int    ssock;   /* Ведомый сокет TCP */
fd set    rfds; /* Дескрипторы, готовые к чтению */
/* инициализация сокетов  tsock и usock */
nfds = MAX(tsock, usock) +1;  /*Длина битовой маски для набора дескрипторов */
while (1)  {
    FD_ZERO(&rfds);
    FD_SET(tsock,  &rfds); FD_SET(usock, &rfds);
    if (select(nfds, &rfds,  (fd_set *)0,  (fd_set *)0, (struct timeval *)0)
<0) {/* ошибка */}
    if (FD_ISSET(tsock, &rfds)) {
        len=sizeof(fsin);
    ssock = accept(tsock, (struct sockaddr*)&fsin,&len); if(ssock<0){
/*ошибка*/}
        Handler_tcp(ssock, buf);  /* обработчик tcp клиента основан на
    параллельном потоке или процессе*/
        ……
    }
    if  (FD_ISSET(usock,  &rfds)) {
        len = sizeof(fsin);
        if  (recvfrom(usock, buf,  sizeof(buf), 0, (struct sockaddr *)&fsin,
        &len) <0) {/*ошибка*/}
        ……
        Handler_udp(buf);
    }
```

СПАСИБО ЗА ВНИМАНИЕ

Программа типа клиент-сервер для TCP



Применение процессов для обеспечения параллельной работы сервера

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

Родительский процесс

```
...
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
…
}
If(f>0){
Close(sn);
... }
...}
```

Можно узнать о наличии зомби-процесса командой: `>ps –aux`

Child for client1

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
… }
If(f>0){
}
… }
```

Child for client2

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
… }
If(f>0){
}
… }
```

Child for client3

```
…
for(;;){
sn=accept(s,….);
f=fork();
If(f==0){
Close(s);
Send(sn,…);
Recv(sn,…);
…
Close(sn);
… }
If(f>0){
}
… }
```

Применение потоков для обеспечения параллельной работы сервера