1. Билет №8

Средства взаимодействия процессов — сокеты Беркли. Создание сокета — семейство, тип, протокол. Системный вызов sys_socket() и struct socket. Состояния сокета. Адресация сокетов и ее особенности для разных типов сокетов. Модель клиент-сервер. Сетевые сокеты — сетевой стек, аппаратный и сетевой порядок байтов. Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов и мультиплексированием (лаб. раб.).

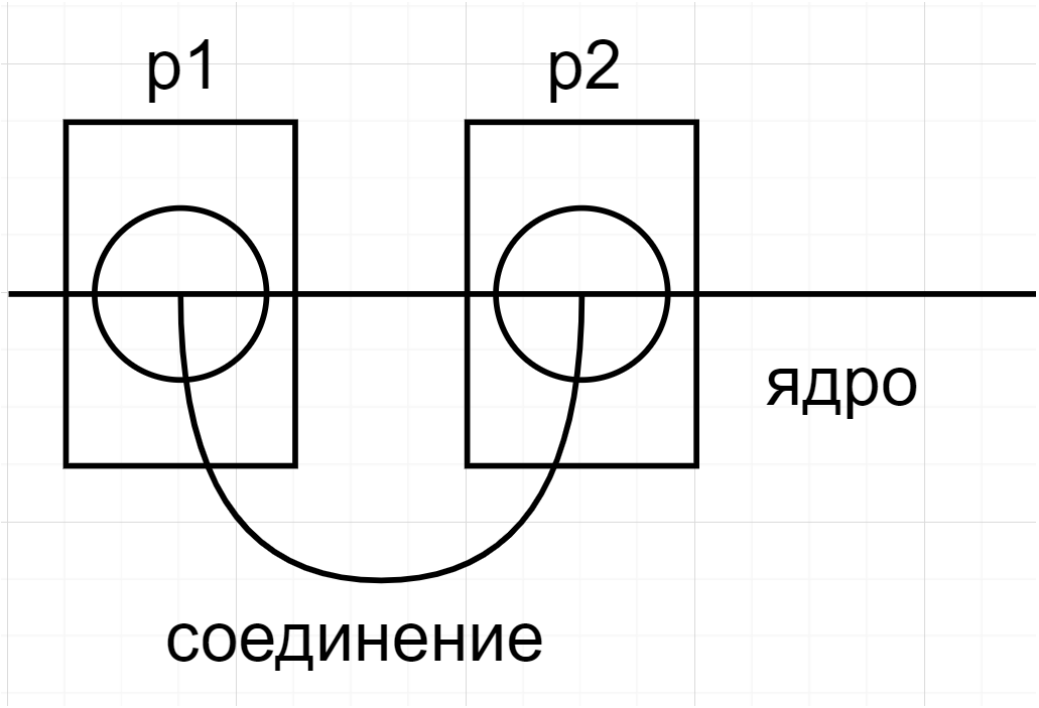
1.1. Средства взаимодействия процессов — сокеты Беркли

Сокеты – универсальное средство взаимодействия параллельных процессов. Универсальность заключается в том, что сокеты используются как и на локальной машине, так и в распределенной системе (сети), в отличие от, например, разделяемой памяти, которая применима только на отдельно стоящей машине.

Распределенная система — система с раздельной памятью (объединенные в сеть компы с собственной памятью, в наших компах все ядра работают с общей памятью).

Сокет – абстракция конечной точки соединения.

Взаимодействие на отдельной машине и в сети существенно разное: в сети это будет транспортный уровень (сетевой протокол, например, TCP/IP)



Парные сокеты обеспечивают дуплексную связь, т.е. сообщения можно передавать через один сокет в обе стороны (альтернатива pipe)

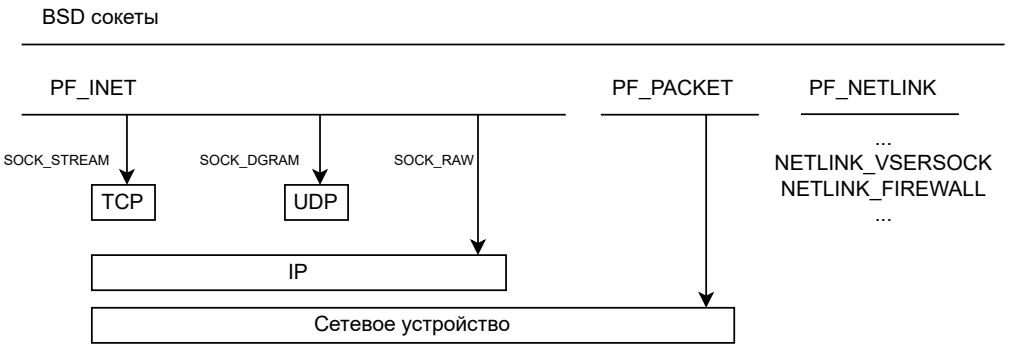
Парные сокеты vs программные каналы

Парные сокеты были созданы в UNIX BSD как универсальное (могут быть использованы для взаимодействяи параллельных процессов на отдельно стоящей машине и в распределенных системах) средство взаимодействия параллельных процессов.

Распределенная система - у каждого узла (хоста) своя память.

Отличия от pipe: парные сокеты обеспечивают дуплесную связь (двусторонюю, читение и запись), а pipe - симплексную (односторонюю)

BSD Сокеты



Сокеты Packet созданы для непосредственного доступа приложений е сетевым устройстам

Сокетов Netlink очень много, основные: NETLINK_USERSOCK, NETLINK_FIREWALL.

Созданы для обмена данными между частями ядра и пространством пользователя.

Связь виртуальной файловой системы proc и сокетов NETLINK

В Linux есть ВФС proc, созданная специально для того, чтобы в пространстве пользователя можно было получить информацию о выполнении процессов. Но в ядре информации значительно больше (и о процессах, и о ресурсах). Очень важно иметь возможность получить ее. Ядро предоставляет средства для получения этой информации. Одним из таких средств являются сокеты NETLINK.

1.2. Создание сокета

Сокеты для взаимодействия на отдельно стоящей машине/ в сети создаются системным вызовом

```
1 int socket(int family, int type, int protocol);
```

Синопсис — краткое описание. Не называть это сигнатурой!

Параметры системного вызова socket()

- family/domain - пространство имен
  - AF_UNIX - межпроцессорное взаимодействие на отдельно стоящей машине, часто говорят “домен UNIX”. Сокеты в файловом пространстве имен.
  - AF_INET - семейство TCP/IP для интернета версии 4 (IPv4). Интернет-домен, фактически любая компьютерная сеть
  - AF_INET6 - семейство TCP/IP для IPv6
  - AF_IPX - домен протокола IPX
  - AF_UNSPEC - неопределенный домен

AF - address family. Сокеты на отдельно стоящей машине (локальные сокеты) взаимодействуют через файловое пространство имен. Чтобы организовать взаимодействие процессов через сокеты AF_UNIX, объявляется файл, который виден в файловой подсистеме как специальный файл (s - маленькая)

- SOCK_STREAM - потоковые сокеты. Определяет ориентированное на потоки, надержное, упорядоченное, логическое соединение между двумя сокетами
- SOCK_DGRAM - опредялеют ненадежную службу дейтаграм без установления логического соединения, где пакеты модут передаваться без сохранения порядка (широковещательная передача данных)
- SOCK_RAW - низкоуровневые сокеты

- protocol
  обычно ставится 0 - протокол назначается по умолчанию. Например, для AF_INET SOCK_STREAM протокол TCP, для SOCK_DGRAM это UDP, но можно задать протокол предописанной константой IPPROTO_*, например, IPPROTO_TCP

API — прикладной программный интерфейс, предоставляемый ОС (для программиста: набор функций, которые можно вызвать из приложения). Application Program Interface. API-функция => режим ядра (системный вызов).

1.3. Системный вызок sys_socket()

В ядре socket вызывает sys_socket (листинг sys_socket чуть ниже).

На лекциях не было, кусок кода ядра.

```
1 SYSCALL_DEFINE3(socket, int, family, int, type, int, protocol)
2 {
3   return __sys_socket(family, type, protocol);
4 }
```

```
1 #include <net/socket.c>
2 asmlinkage long sys_socketcall(int call, unsigned long *args)
3 // ее текст = switch, переключающий ядро на разные функции, связанные с со
      кетом
4 {
5   int err;
6   if copy_from_user(a, args, nargs[call])
7     return –EFAULT;
8   a0 = a[0];
```

```
9    a1 = a[1];
10   switch(call)
11   {
12       case SYS_SOCKET: err= sys_socket(a0, a1, a[2]); break;
13       case SYS_BIND: err= sys_bind(a0, (struct sockaddr*)a1, a[2]); break;
14       case SYS_CONNECT: err= sys_connect(...); break;
15       ...
16       default: err = -EINVAL; break;
17   }
18   return err;
19 }
```

В switch перечисляются функии так называемого сетевого стека. Для них определены предописанные константы (макроопределения): (код с дефайнами относится к пояснению)

```
1 <include/linux/net.h>
2 #define SYS_SOCKET 1
3 #define SYS_BIND 2
4 #define SYS_CONNECT 3
5 #define SYS_LISTEN 4
```

```
1 asmlinage long sys_socket(int family, int type, int protocol)
2 {
3   int retval;
4   struct socket *sock;
5   ...
6   retval = sock_create(famaly, type, protocol, &sock);
7   ...
8   return retval;
9 }
```

1.4. struct socket. Состояния сокета

```
1 struct socket // нет в 6 версии ядра
2 {
3   socket_state state;
4   short type;
5   unsigned long flags;
```

```
6     const struct proto_ops *ops;
7     struct fasync_strcut *fasync_list;
8     struct file *file;
9     struct sock *sk;
10    wait_queue_head_t wait;
11  }
```

flags - используется для синхронизации доступа.

struct proto_ops - действия на сокете (protocol operations). Здесь можно зарегистрировать свои функции работы с сокетами.

У сокета различают 5 состояний, 4 из которых - стадии соединения:

- SS_FREE - свободный сокет, с которым можно соединяться;
- SS_UNCONNECTED - несоединенный сокет;
- SS_CONNECTING - сокет находится в состоянии соединения;
- SS_CONNECTED - соединенный сокет;
- SS_DISCONNECTING - сокет разъединяется в данный момент.

Сокеты описываются как открытые файлы (они не хранятся во вторичной памяти, это файлы специального типа (сможем увидеть только в сокетах в файловом пространстве имент AF_UNIX))

1.5. Адресация сокетов и ее особенности для разных типов сокетов

struct sockaddr - обращение к сокету выполняется по адресу (сокеты адресуются)

Взаимодействие на сокетах происходит по модели клиент-сервер

Адресация сокетов:

```
1  struct sockaddr
2  {
3    sa_family_t sa_family;
4    char sa_data[14];
5  }
```

Такая структура адреса не подходит для интернета, так как там необходимо указывать номер порта и сетевой адрес. Для интернета разработана другая структура:

```
1 struct soackaddr_in
2 {
3   sa_family_t sa_family;
4   unsigned short int sin_port;
5   struct in_addr sin_addr;
6   unsigned char sin_zero[sizeof(struct sockaddr) - sizeof(sa_family_t) -
        sizeof(uint16_t) - sizeof(struct in_addr)];
7 };
```

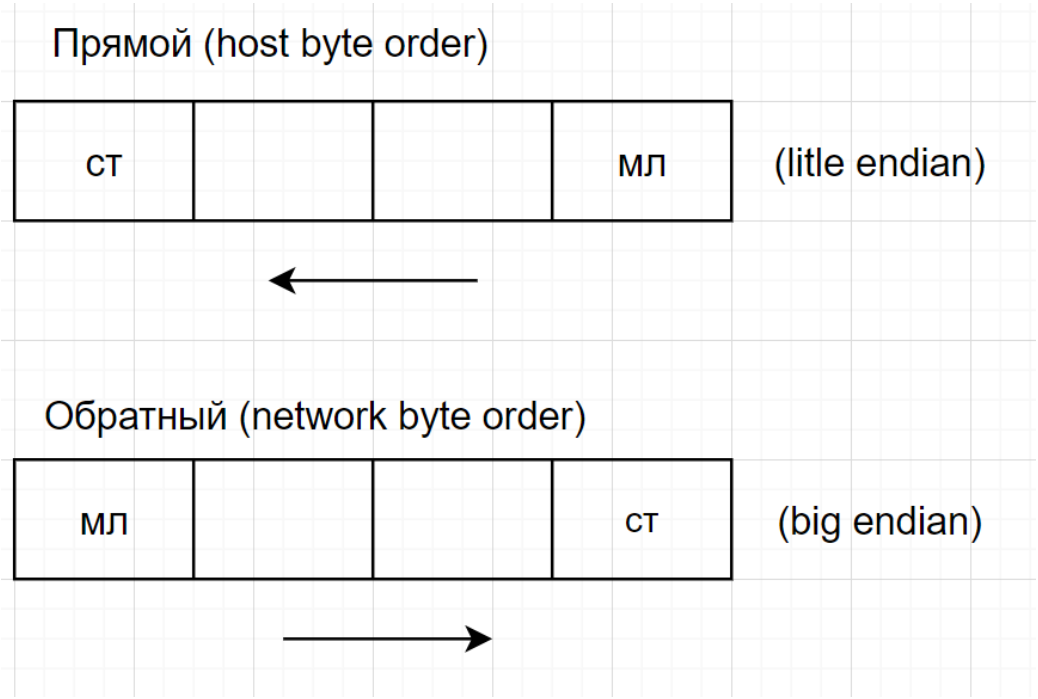
Сетевой адрес и номер порта должны быть указаны в сетевом порядке байтов.

1.6. Аппаратный и сетевой порядок байтов

Порядок байт:

- аппаратный
- сетевой

Прямой и обратный порядок байт



Сети оперируют портами и сетевыми адресами

```
1 uint16_t htons(uint16_t hostint16) // host to network short
2 uint32_t htonl(uint32_t hostint32) // host to network long
3
4 uint16_t ntohs(uint16_t netshort); // network to host short
5 uint32_t ntohl(uint32_t netlong); // network to host long
```

1.7. Модель клиент-сервер

Взаимодействие на сокетах осуществляется по модели клиент-сервер: сервер предоставляет ресурсы и службы одному или нескольким клиентам, которые обращаются к серверу за обслуживанием.

В момент, когда клиент запрашивает соединение с сервером, сервер может либо принять, либо отклонить запрос на соединение. Если соединение устанавливается, то оно поддерживается по определённому протоколу.

1.8. Сетевой стек

Сети распределенные системы, т.е. у каждого хоста своя память

В сетях - только передача сообщений, которые должны сопровождаться адресом

Пакет - сообщение с адресом + служебная информация

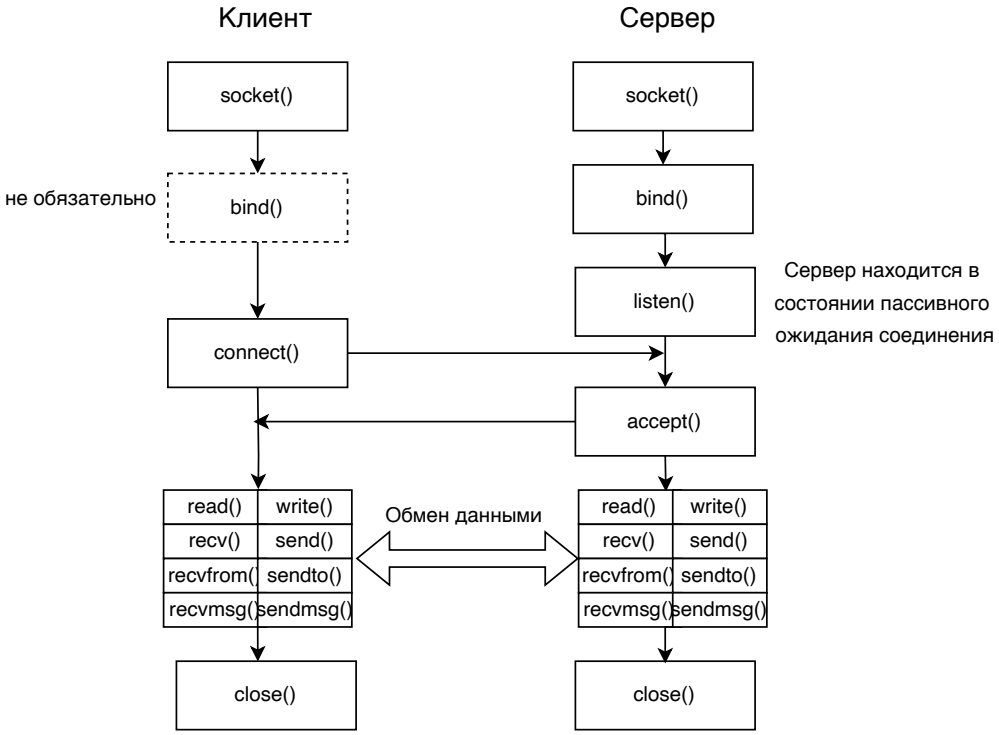
В Linux определен интерфейс между пользовательскими процессами и стеком сетевых протоколов в ядре.

Это не по семинару*

Модули протоколов группируются по семействам протоколов, такими, как AF_INET, AF_IPX и AF_PACKET, и типам сокетов, такими, как
SOCK_STREAM или SOCK_DGRAM. Сетевой стек ядра Linux имеет две структуры:

struct socket — интерфейс высокого уровня, который используется для системных вызовов (именно поэтому он также имеет указатель struct file, который представляет файловый дескриптор)

struct sock — реализация в ядре для AF_INET сокетов (есть также struct unix_sock для AF_UNIX сокетов, которые являются производными от данного), которые могут использоваться как в ядре, так и в режиме пользователя.



socket() - создание точки соединения. Возвращает файловый дескриптор. Сокет - специальный файл (у него есть inode), назначение которого - обеспечивать соединения;

AF_INET, SOCK_STREAM - сетевое взаимодействие по протоколу TCP

bind() связывает сокет с адресом (сетевым (порт + API-адрес) в случае сокетов AF_INET)

```
1 int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

struct sockaddr_in - есть поле “порт” и “сетевой адрес” (у них должен быть сетевой порядок (применяем функцию htons())

На сервере вызов bind() обязателен, на клиенте нет, т.к. его точный адрес часто не играет никакой роли (если bind() не вызывается, адрес назначается клиентам автоматически)

listen() информаирует ОС о том, что он готов принимать соединения (имеет смысл только для протоколов, ориентированных на соединение (например, TCP)

```
1 int listen(int sockfd, int baclog);
```

connect() - клиент устаналивает активное соединение с сокетом (с сервером)

```
1 int connect(int sockfd, struct sockaddr *addr, int addrlen)
```

Для протокола без соединения (например, UDP) connect может использоваться для указания адреса назначения всех передаваемых пакетов

accept() - вызывается на стороне сервера, если соединение установлено. Сервер принимает соединение, *только если* он получил запрос на соединение.

```
1 int accept(int sockfd, void* addr, int *addrlen)
```

Когда соединение принимается, accept() создает копию исходого сокета, чтобы сервер мог принимать другие соединения. Исходный сокет остается в состоянии listen, а копия будет находиться в состоянии connected. accept() возвращает файловый дескриптор копии исходного сокета.

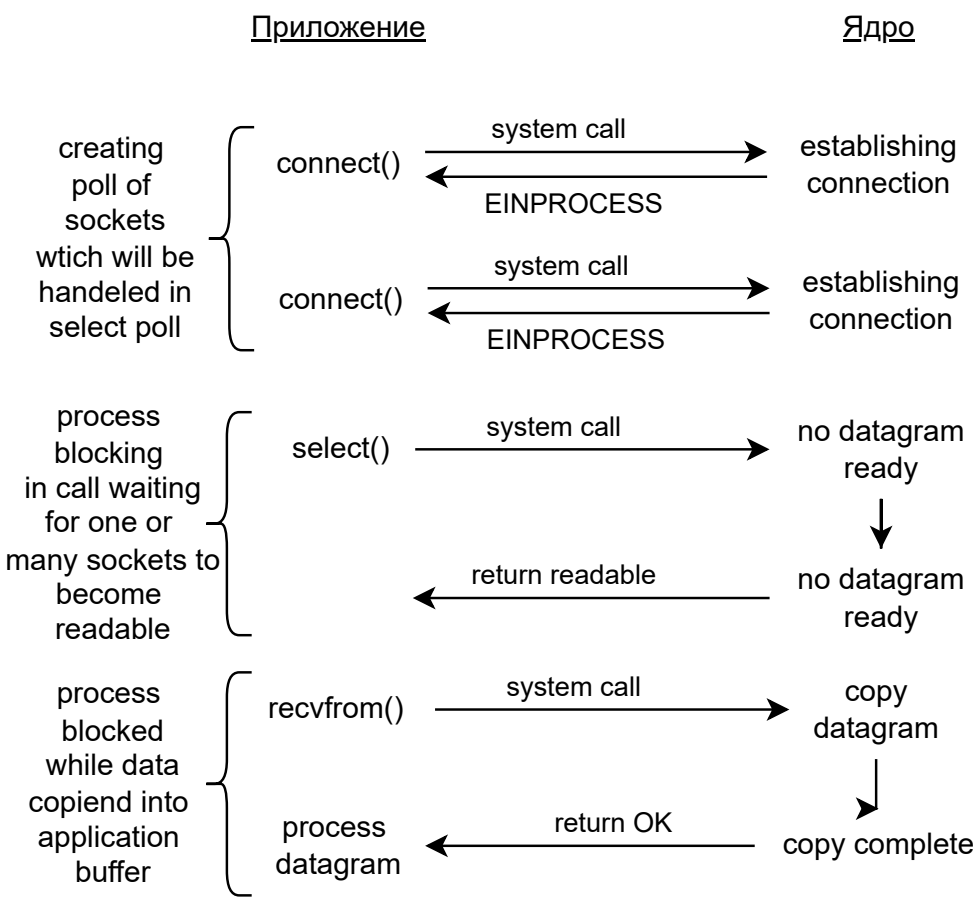
Про уровни сетевых протоколов

Протоколы различаются по уровням. Нижний уровень - непосредсвенное взаимодействие с аппаратной частью (самое важное)

1.9. Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов и мультиплексированием (лаб. раб.)

Сетевые сокеты с мультиплексированием:

Мультиплексирование - альтерната многопоточности (созданию дочернего процесса/-потока для обработки каждого соединиия)



Это детализированная схема: клиенты вызывают connect() и создается пул сокетов.

Для сокращения времени блокировки сервера в ожидании соединения используется select() (пока соединение не возникнет, сервер будет блокирован на accept(), т.е. будет в состоянии пассивного ожидания соединения), т.к. время установления соединения со многими клиентами меньше, чем с каждым конкретным клиентом в определенной последовательности.

В результате select() создает пул соединение. Есть макрос, который “реагирует” на возникновение хотя бы одного соединения. В результате будет вызан accept(), который последовательнос принимает соединения.

Для создания пула соединений можно использовать массив.

Мультиплексор опрашивает соединения. Когда соединение готово, оно фиксируется ядром.

Мультиплексоры: select pool pselect epool

Код клиента

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#define SERVER_PORT 8080
#define MSG_LEN 64

int main(void)
{
  setbuf(stdout, NULL);

  struct sockaddr_in serv_addr =
  {
    .sin_family = AF_INET,
    .sin_addr.s_addr = INADDR_ANY,
    .sin_port = htons(SERVER_PORT)
  };
  socklen_t serv_len;

  char buf[MSG_LEN];

  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  // error handling

  if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
      0)
    // error handling

  char input_msg[MSG_LEN], output_msg[MSG_LEN];
  sprintf(output_msg, "%d", getpid());

```

```
38    if (write(sock_fd, output_msg, strlen(output_msg) + 1) == -1)
39      // error handling
40
41    if (read(sock_fd, input_msg, MSG_LEN) == -1)
42      // error handling
43
44    printf("Client receive: %s \n", input_msg);
45    close(sock_fd);
46    return EXIT_SUCCESS;
47  }
```

epoll_fd — ФД, который описывает новый еполл объект, нужен для всех вызовов интерфейса еполл (API мультиплексированоного в/в).

bind() - связывает сокет с заданным адресом (для AF_UNIX с файлом). После вызова bind() программа-сервер становится доступна для соединения по заданному адресу (имени файла)

struct sockaddr — структура адреса.

O_NONBLOCK — сокет открыт в неблокирующем режиме (при маленьком размере буфера и большом размере пакета будет большое количество вызовов select(), чтобы вызвать его 1 раз, нужен неблокирующий режим).

EPOLLIN — событие активно, когда есть данные на вход.

EPOLLOUT — событие активно, когда есть данные на выход.

EPOLLET (edge-triggered) — включает прерывание по фронту (при добавлении в epoll слушающего сокета нужно оставить только флаг EPOLLIN). Прерывание по фронту разблокирует epoll_wait (срабатывает) только когда меняется состояние события. Прерывание по уровню срабатывает все время, пока событие находится в требуемом состоянии. Прерывание по уровню аналогично обычному pool/select. Для EPOLLET сокет должен быть открыт в неблокирующем режиме (если спросит где это видно в коде - где-то написано O_NONBLOCK).

По сути, без этого флага разблокировка epoll будет осуществляться каждый раз, когда есть необработанное событие, то есть если в сокете появились данные, доступные для чтения, и не было произведено чтения, то при следующем вызове epoll_wait снова произойдет разблокировка. В случае установки EPOLLET разблокировка при повторном epoll_wait не произойдет, даже если данные не были прочитаны на прошлой итерации. Флаг EPOLLET это то, что делает epoll O(1) мультиплексором для событий — очень быстро.

setsockopt(sock, SOL_SOCKET,
SO_REUSEADDR, &sopt, sizeof(sopt) — установка опций сокета: SOL_SOCKET — нужен

для манипуляции флагами сокета, SO_REUSEADDR — повторное использование адреса после вызова accept. sopt — заполняется.

epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sock, &epev) — связываем объект епола с событием, EPOLL_CTL_ADD — добавление события.

epoll_wait(epoll_fd,&epev,
CLIENTS_MAX+1, 1) — опрос.

Что за файловый дескриптор в accept? Исходный. Accept возвращает копию ФД исходного сокета. Копия будет в состоянии connected. Исходный сокет остается в состоянии listen. Копия исходного сокета добавляется в пул соединений.

```
1  int accept(int sockfd, struct sockaddr *_Nullable restrict addr,
2                 socklen_t *_Nullable restrict addrlen);
```

Что за проверка, почему ветвимся *показывает на схему*. Проверка на то, что созданный сокет является исходным.

Код сервера

```
1  #include <arpa/inet.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/epoll.h>
5  #include <sys/socket.h>
6  #include <unistd.h>
7
8  #define CLIENTS_MAX 5
9  #define PORT 8181
10 #define BUF_SIZE 128
11
12 int main()
13 {
14   int epoll_fd;
15
16   if ((epoll_fd = epoll_create(CLIENTS_MAX)) == -1)
17   {
18     perror("Can't epoll_create");
19     exit(1);
20   }
21
22   int sock;
23
```

```
24  if ((sock = socket(AF_INET, SOCK_STREAM | O_NONBLOCK, 0)) == -1)
25  {
26    perror("Can't socket");
27    exit(1);
28  }
29
30  int sopt = 1;
31
32  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sopt, sizeof(sopt)) ==
        -1)
33  {
34    perror("Can't setsockopt");
35    exit(1);
36  }
37
38  struct sockaddr_in addr;
39
40  addr.sin_family = AF_INET;
41  addr.sin_addr.s_addr = INADDR_ANY;
42  addr.sin_port = htons(PORT);
43
44  if (bind(sock, (struct sockaddr *) &addr, sizeof(addr)) == -1)
45  {
46    perror("Can't bind");
47    exit(1);
48  }
49
50  if (listen(sock, CLIENTS_MAX) == -1)
51  {
52    perror("Can't listen");
53    exit(1);
54  }
55
56  struct epoll_event epev;
57
58  epev.events = EPOLLIN;
59
60
```

```
61    epev.data.fd = sock;
62
63    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sock, &epev) == -1)
64    {
65      perror("Can't epoll_ctl");
66      exit(1);
67    }
68
69    while (1)
70    {
71      struct epoll_event epev[CLIENTS_MAX + 1];
72      int num;
73
74      if ((num = epoll_wait(epoll_fd, &epev, CLIENTS_MAX + 1, -1)) == -1)
75      {
76        perror("Can't epoll_wait");
77        exit(1);
78      }
79
80      for (int i = 0; i < num; i++)
81      {
82        if (epev[i].data.fd == sock)
83        {
84          int conn;
85
86          if ((conn = accept(sock, NULL, NULL)) == -1)
87          {
88            perror("Can't accept");
89            exit(1);
90          }
91
92          struct epoll_event epev;
93
94          int flags = fcntl(conn, F_GETFL, 0);
95          fcntl(conn, F_SETFL, flags | O_NONBLOCK);
96
97          epev.events = EPOLLIN | EPOLLET ;
98          epev.data.fd = conn;
```

```
99
100         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn, &epev) == -1)
101         {
102           perror("Can't epoll_ctl");
103           exit(1);
104         }
105       }
106
107       else
108       {
109         int conn = epev[i].data.fd;
110
111         char received_msg[BUF_SIZE], send_msg[BUF_SIZE];
112
113         if (recv(conn, received_msg, sizeof(received_msg), 0) == -1)
114         {
115           perror("Can't recv");
116           exit(1);
117         }
118
119         printf("Server %d received message: %s\n", getpid(), received_msg)
              ;
120         sprintf(send_msg, "%s from server with pid %d", received_msg,
              getpid());
121
122         if (send(conn, send_msg, sizeof(send_msg), 0) == -1)
123         {
124           perror("Can't send");
125           exit(1);
126         }
127
128         printf("Server %d send message: %s\n", getpid(), send_msg);
129         close(conn);
130       }
131     }
132   }
133
134   return 0;
```

```
135 }
```