Билет 3 **3**

Билет 7 **3**

Билет 7 старый. **3**

Билет 8 **3**

Билет 8 старый **4**

Модели ввода-вывода: представление с помощью диаграмм, описание и особенности. 5

Блокирующий ввод-вывод (blocking I/O) (синхронный) 5

Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод (диаграмма). 6

Неблокирующий ввод-вывод (nonblocking I/O) (характерно для сокетов) 8

Ввод-вывод с мультиплексированием (I/O multiplexing: select and poll) 9

Ввод-вывод, управляемый сигналом (signal driven I/O: SIGIO) 12

Асинхронный ввод-вывод (asynchronous I/O: the POSIX aio functions) 13

Классификация моделей ввода-вывода 14

Средства взаимодействия процессов — сокеты Беркли. 17

Создание сокета — семейство, тип, протокол. 17

1.domain или family 18

2.тип сокета (type) 19

3.Protocol 20

Ошибки 20

sys_socketcall 21

Системный вызов sys_socket() 22

struct socket – структура, описывающая сокет 22

Состояния сокета. 23

Адресация сокетов и ее особенности для разных типов сокетов. 23

Форматы адресов 23

Связь сокета с адресом 25

Аппаратный и сетевой порядок байтов. 26

Пример из лабы 26

Модель клиент-сервер. 28

Сетевые сокеты — сетевой стек. 29

Сетевой стек 29

сетевые сокеты 31

Сокеты AF_UNIX =Сокеты в файловом пространстве имен 34

Примеры 34

Мультиплексирование 36

Способ 1 36

Способ 1_2 38

Способ 2 38

Что такое неблокирующие сокеты 39

Способ 2_1: polling 39

Способ 2_2: select (и современная версия pselect()) 39
Способ 2_3: poll (и современная версия epoll()) (насколько я понимаю, техника polling вообще не связано с poll()) 42
Как представляются дескрипторы 42
Сами функции 42
epoll 43
Примеры из лабораторной работы: 44
select (мультиплексирование); AF_INET, SOCK_STREAM 44
epoll (мультиплексирование): на всякий 50
AF_UNIX, SOCK_DGRAM; с отправкой ответа от сервера клиенту, 2 сокета 56
AF_UNIX, SOCK_DGRAM; без отправки ответа от сервера клиенту 61
AF_UNIX, SOCK_STREAM. Криво, плохо, не повторять!: своего рода мультиплексирование 64
AF_UNIX, SOCK_STREAM | SOCK_NONBLOCK: То же, что и в предыдущем, но еще и с флагом SOCK_NONBLOCK 69

OFFTOP **75**
Сравнение семейств сокетов 75
В целом: функции, определенные на сокетах 75
Что еще 78

Все про (аппаратные) прерывания: **79**
Запрос прерывания и линии IRQ 79
ЧТО ТАКОЕ ЛИНИЯ IRQ 80
Адресация аппаратных прерываний в з-р (трехшинная архитектура). 80
Таблица дескрипторов прерываний (IDT) 82
Типы шлюзов. 83
Формат дескриптора прерывания 84
Пример заполнения IDT из лабораторной работы. 85

- ВВ = ввод-ввывод
- некоторые примеры помечены курсивом (то есть как не очень важное). Это действительно кажется мне не очень важным. Те примеры, которые просятся ПО ЗАДАНИЮ так помечены точно не будут, так что можно спокойно пропускать помеченные курсивом
- есть одна зацикленность ссылок: про мультиплексирование. Грубо говоря, 1 часть – где описывается мультиплексирование как одна из моделей ВВ (и тут идея, все в общих словах), 2 часть – более относящаяся к сокетам (ближе к концу файла) (тут уже про конкретные функции и примеры). Так вот, эти части дают ссылки друг на друга. Я бы по-хорошему ВЕЗДЕ писала ВСЕ, но если будет орать, что мы много всего строчим, то только соответствующую часть))
- ближе к концу файла OFFTOP - чего напрямую не просят ни в одном билете, но на всякий))))()

Билет 3

Модели ввода-вывода: представление с помощью диаграмм, описание и особенности.

Классификация моделей ввода-вывода.

модель клиент-сервер (в билете этого нет отдельным пунктом, но 100% нужно описать, так как сетевой стек ровно об этом)

Сетевой стек.

Мультиплексирование при взаимодействии процессов в распределенных системах по модели клиент-сервер.

Примеры мультиплексоров и пример из лабораторной работы.

Билет 7

Классификация типов ввода-вывода с точки зрения программиста: диаграммы последовательности действий для каждого типа ввода-вывода и описание.

Классификация моделей ввода-вывода.

Особенности и назначение асинхронного ввода-вывода. (про назначение разве что своими словами)

Сетевой стек.

Мультиплексирование.

Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM. Пример (лаб. раб.)

Билет 7 старый.

Классификация типов ввода-вывода с точки зрения программиста: описание и диаграммы последовательности действий для каждого типа ввода, вывода.

Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM. Пример (лаб. раб.)

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.).

Билет 8 старый

Средства взаимодействия процессов-сокеты Беркли.

Типы сокетов.

Адресация.

Сокеты AF_UNIX.

Сетевые сокеты — сетевой стек,

аппаратный и сетевой порядок байтов.

Примеры реализации взаимодействия процессов по модели клиент-сервер с использованием сокетов (лабораторная работа).

Модели ввода-вывода: представление с помощью диаграмм, описание и особенности.

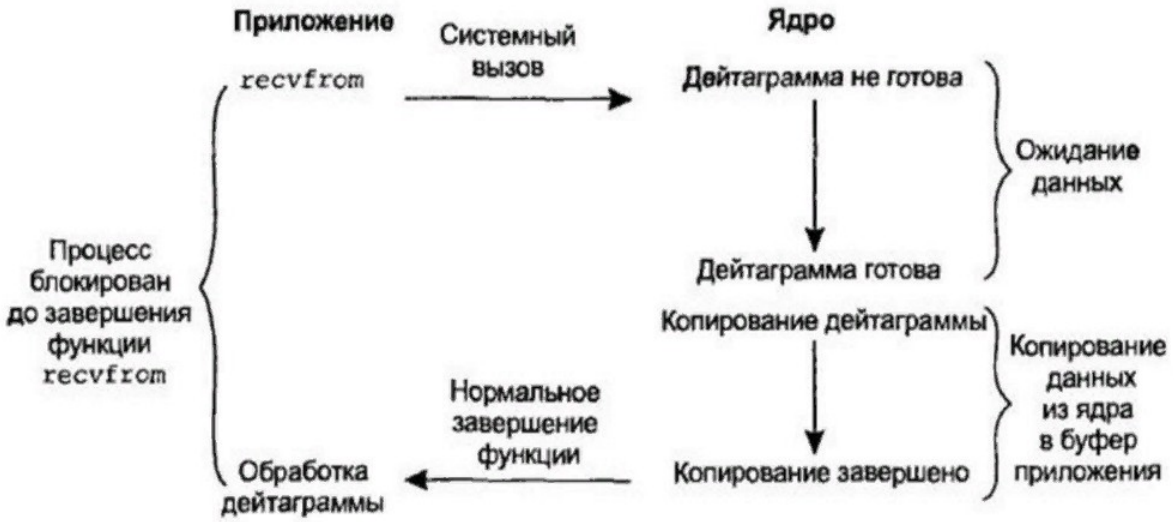
В Unix/Linux поддерживается 5 (+½) моделей ввода вывода (с точки зрения программиста):

- блокирующий ввод-вывод (blocking I/O)
- неблокирующий ввод-вывод (nonblocking I/O)
- ввод-вывод с мультиплексированием (I/O multiplexing: select and poll)
- ввод-вывод, управляемый сигналом (signal driven I/O: SIGIO)
- асинхронный ввод-вывод (asynchronous I/O: the POSIX aio functions)

Обычно есть две отдельные фазы для операции ввода:

1. Ожидание готовности данных. (Этот шаг в случае операции ввода в сокет обычно включает ожидание поступления данных в сеть; когда пакет прибывает, он копируется в буфер в ядре).
2. Копирование данных из ядра в буфер процесса

1. Блокирующий ввод-вывод (blocking I/O) (синхронный)



Самая распространенная модель.

При блокирующем вводе-выводе процесс блокирован все время от момента, когда был выполнен вызов recvfrom, до момента, когда дейтаграмма поступает и копируется в буфер приложения. Системный вызов может передать в буфер приложения данные или ошибку. При успешном завершении вызова приложение обрабатывает полученные данные.

Offtop: Почему отдельно указано, что он именно синхронный? Есть тема: синхронный и асинхронный ВВ. Подчеркивается, что асинхронный ВВ невозможен для обычных файлов. Там реализовываются соответствующие флаги, всегда будут блокировки (а блокировки - это зло). Время нельзя предсказать – они случайны. Мьютексы надо использовать только там, где это необходимо.

На экзамене надо еще (с прошлого года):

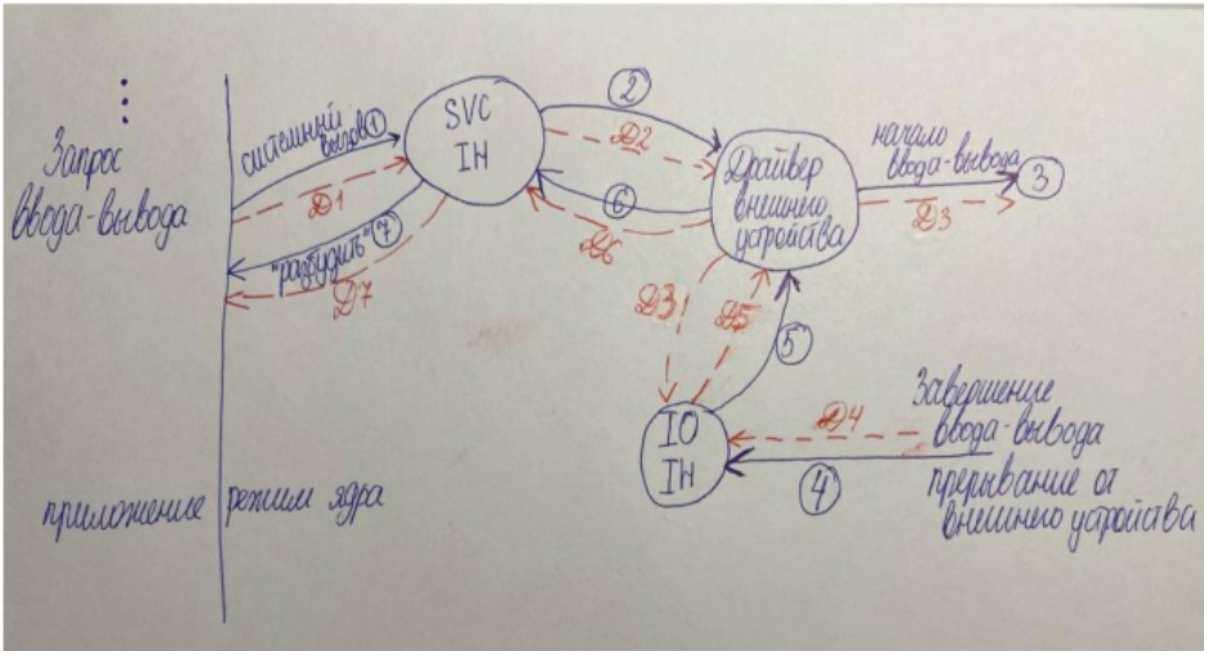
Прерывания в последовательности ввода-вывода — обслуживание запроса процесса на ввод-вывод (диаграмма).

(вроде это называется диаграммой Шоу)

Основную группу прерываний составляют прерывания от устройств ВВ. Без устройств ВВ пользоваться вычислительной системой невозможно, тк именно они предназначены для взаимодействия с пользователем.

SVC — supervisor call
IH — interrupt handler
Супервизор — ОС в стадии выполнения

Последовательность действий в системе при запросе приложения на ввод/вывод:



Любой запрос ввода/вывода блокирует процесс на какое-то время (даже open()!). Здесь происходит вызов read/write и система переходит в режим ядра (ни одна система не дает прямое обращение к внешним устройствам)

Действия (красным на схеме):

(обратить внимание, что у Д3 две стрелочки)

(на лекции все писалось сплошным текстом, но вроде если раскидать по действиям, то получится так)

1. Системный вызов из функции, который переводит систему в режим ядра, туда же соответствующие данные.
2. В результате обработки системного вызова будет вызван драйвер (одна из его точек входа) внешнего устройства (программа ввода/вывода). Драйверу будут переданы данные в его формате
3. Драйвер инициализирует работу внешнего устройства, передает по шине данных в контроллер устройства данные в формате, «понятном» устройству (кавычки обязательны!!!!). На этом управление процессором работой заканчивается, он отключается, потому что работой внешнего устройства управляют контроллеры.
4. По завершении операции ввода/вывода данные от контроллера поступают в регистр данных контроллера прерываний и находятся там пока не сработает цепочка обработки прерывания. Контроллером устройства будет сформировано прерывание, которое в простой схеме поступит на контроллер прерывания (IO IH) и в результате будет определен адрес точки входа обработчика прерывания.

5. Процессор перейдет на выполнение обработчика, тк аппаратные прерывания имеют наивысший приоритет. **Обработчики прерываний** входят в состав драйвера и **является одной из точек входа драйвера** внешнего устройства. Драйвер всегда содержит 1 обработчик прерывания.
6. Поскольку обработчик – точка входа драйвера, у драйвера есть call back функция – задача вернуть запрашиваемые данные приложению. В результате драйвер через подсистему ввода/вывода должен передать данные приложению.

7. Для этого работа приложения дб возобновлена (разбудить). Процесс, который запросил вв разблокируется.

С монитором мы работаем как с памятью – mov. А input/output – использование команд ввода вывода, и это приводит к блокировкам.

Возникновение прерывания происходит асинхронно. Чтобы получить значение, процесс разблокируется. Поэтому эта схема называется блокирующий синхронный ввод-вывод

Это возможно в архитектуре, где реализовано распараллеливание функций: канальная (используются каналы) и шинная (контроллеры или адаптеры) архитектуры – устройствами управляют устройства.

Даже если при выполнении ВВ не удалось прочитать или записать данные, приложение все-равно получит информацию (об ошибке)

для информирования процессора о завершении операции ввода-вывода контроллер внешнего устройства формирует сигнал, который по линии IRQ№ поступает на контроллер прерывания; контроллер формирует сигнал прерывания, который по

линии шины данных поступает на выделенный pin процессора; в конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания и при его поступлении переходит на выполнение обработчика данного прерывания; обработчик прерывания должен сохранить данные, пришедшие от контроллера устройства, в буфере ядра, чтобы затем функции обратного вызова драйвера устройства доставили полученные данные в буфер приложения. Для этого приложение разблокируется.

(и если захочется еще про адресацию рассказать – смотри конец файла)

2. Неблокирующий ввод-вывод (nonblocking I/O) (характерно для сокетов)

Неблокирующий ввод-вывод не ожидает наличия данных (или возможности вывода), а получает результат выполнения операции или невозможность ее выполнения в данный момент, что определяется по коду возврата.

Первые три раза системный вызов на чтение данные не возвращает, а возвращает ошибку EWOULDBLOCK. Следующий системный вызов выполняется успешно, так как данные готовы для чтения. Ядро скопирует данные в буфер приложения и будут обработаны.

Такой режим работы называется опросом (polling), так как выполняется постоянный вызов, в данном случае, recvfrom. Для организации такого опроса может быть создан цикл, вызывающий соответствующий системный вызов. Очевидно, что такой подход приводит к большим накладным расходам (overhead) – пустой трате процессорного

времени. Эта модель иногда встречается, как правило, при работе с внешними устройствами.

Пример неблокирующего ввода:

```
int fo[2]; // pipe-канал для чтения из дочернего процесса
if (pipe (fo)) {
      perror ("pipe");
      exit(EXIT_FAILURE);
}
close (fo[1]);
int cur_flg = fcntl(fo[0], F_GETFL); // чтение должно быть в
режиме O_NONBLOCK
if (fcntl(fo[0], F_SETFL, cur_flg | O_NONBLOCK) == -1) {
      perror ("fcntl");
      exit(EXIT_FAILURE);
}

...

while(1) {
      int n = read(fdi, buf, buflen);
      if (n > 0) {
            // данные считаны ...
            // обработка
      } else if (n == -1) {
            if(EAGAIN == errno) { // данные не готовы
                  printf("not ready!\\n");
                  usleep(300);
            } else {
                  perror("read pipe\\n")
                  exit(EXIT_FAILURE);
            }
      }
}
```

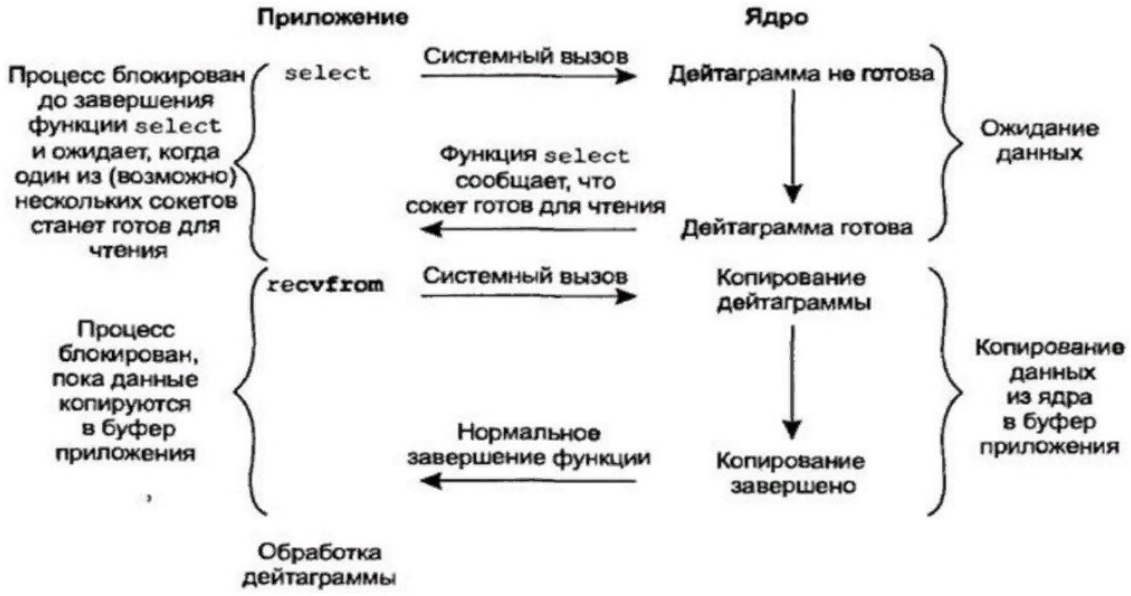
3. Ввод-вывод с мультиплексированием (I/O multiplexing: select and poll)

При мультиплексировании ввода-вывода вызываются мультиплексоры (select, pselect, poll, epoll. Где poll и epoll- более эффективные).

Мультиплексор/коммутатор – устройство, которое объединяет информацию, поступающую по нескольким устройствам ввода и передает ее к одному каналу вывода.

Мультиплексирование (уплотнение) – процесс совмещения нескольких

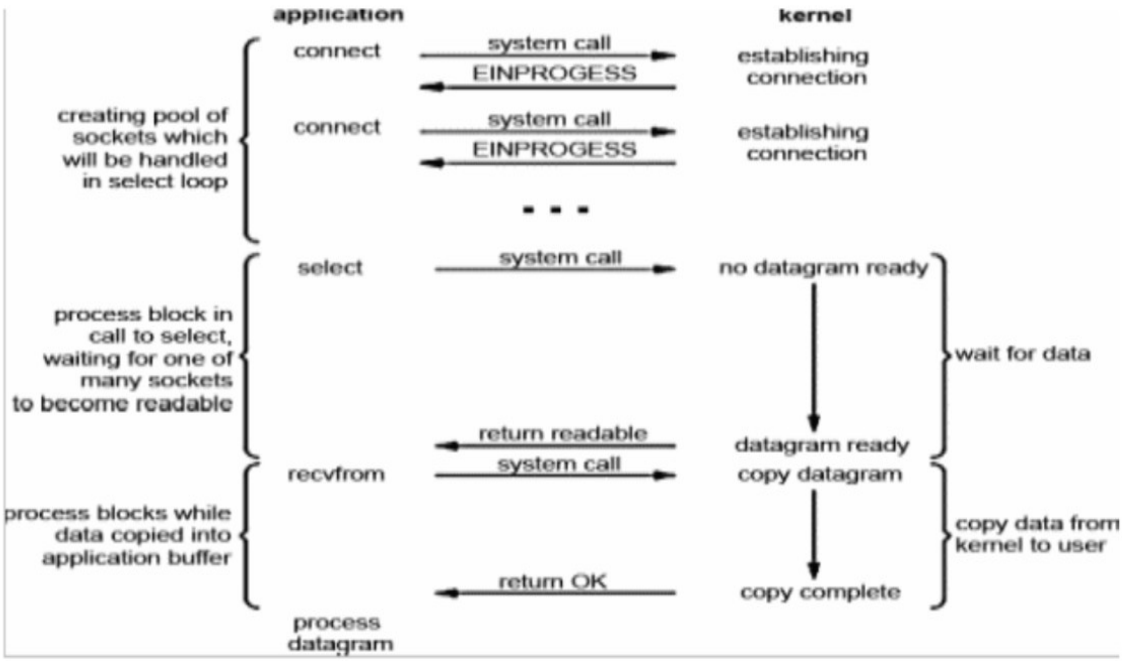
сообщений, передаваемых одновременно в одной физической или логической среде. Существует 2 вида мультиплексирования: временное и частотное



Вместо блокировки в непосредственном системном вызове ввода-вывода процесс блокируется на одном из этих системных вызовов. Приложение блокируется при вызове select, ожидая когда сокет станет доступным для чтения. Затем ядро возвращает приложению статус readable, сообщая, что можно получать данные помощью recvfrom.

Недостаток:
Таким образом, в этой модели есть блокировки, да еще вместо одного – два системных вызова (select и recvfrom), что увеличивает накладные расходы.

Достоинства:
Но в отличии от рассмотренного ранее блокирующего метода, select или любой другой мультиплексор обеспечивает возможность ожидать данные не от одного, а от нескольких файловых дескрипторов, что, очевидно, снижает время блокировки (сна). Это видно из рисунка ниже



Клиенты пытаются создать соединение с сервером, вызывая системный вызов connect. В результате создается пул (pool) дескрипторов сокетов. Получение процессом-клиентом EINPROGRESS означает, что соединение устанавливается.

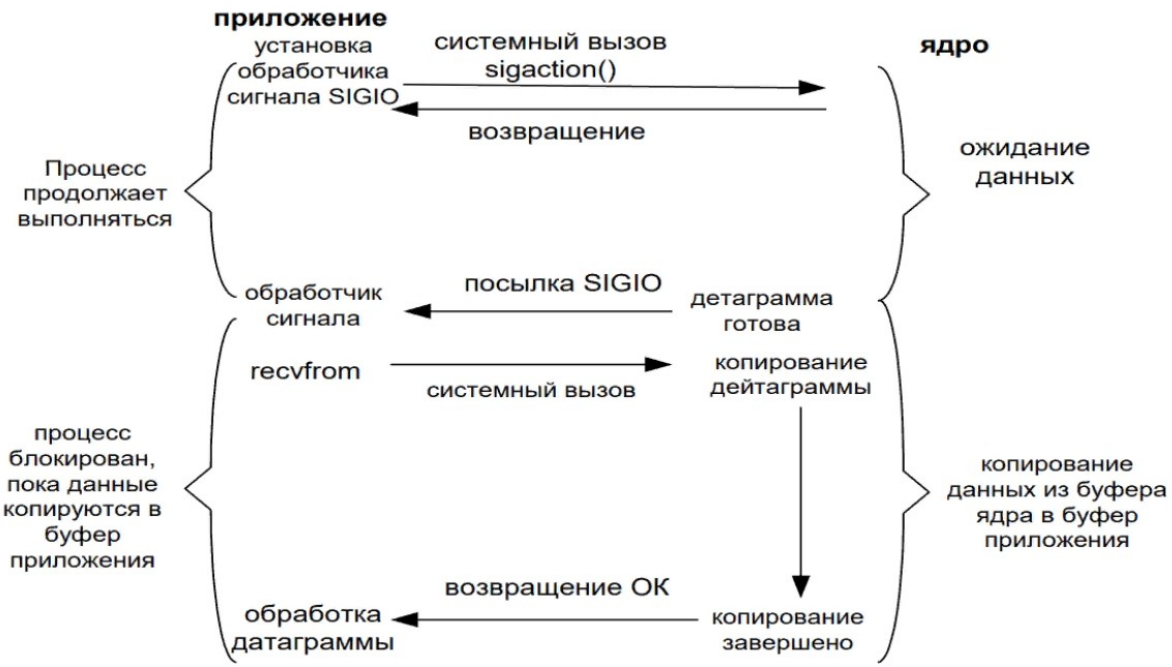
На работу сервера получение клиентом EINPROGRESS никак не влияет, так как мультиплексор обработает первое соединение, которое произойдет: в цикле проверяются все сокеты и берется первый, который готов. Пока соединение принимается (accept), поступают другие соединения. Таким образом снижается время простоя, так как первый раз ожидание может затянуться, но последующие соединения требуют значительно меньших задержек.

Еще один вариант - когда несколько потоков или процессов для обработки запросов к серверу. Создается несколько потоков или процессов, в каждом из которых выполняется блокирующий ввод-вывод. Но следует иметь в виду, что в линукс очень дорогие потоки (большое количество накладных расходов).

мне кажется, что тут главное – описать саму идею мультиплексирования, но если очень хочется, то можно выписать и [подробное описание функций, связанных с этим делом.]

Сравнение мультиплексирования и многопоточности с последних лаб (кидали фотку в группу): Вы должны предпочесть мультиплексирование в потоке, когда количество потоков зависит от количества обслуживаемых вами клиентов.
Также было сказано: мультиплексированием это аналог многопоточности

4. Ввод-вывод, управляемый сигналом (signal driven I/O: SIGIO)



Когда данные готовы к считыванию ядро должно уведомить приложение и послать сигнал SIGIO. При использовании данного способа на сетевом сокете необходимо установить режим ввода-вывода, управляемый сигналом и обработчик сигнала, используя системный вызов sigaction(). Особенностью модели ввода-вывода, управляемого сигналом, является то, что приложение не блокируется, а продолжает работать несмотря на то, что данные не готовы.

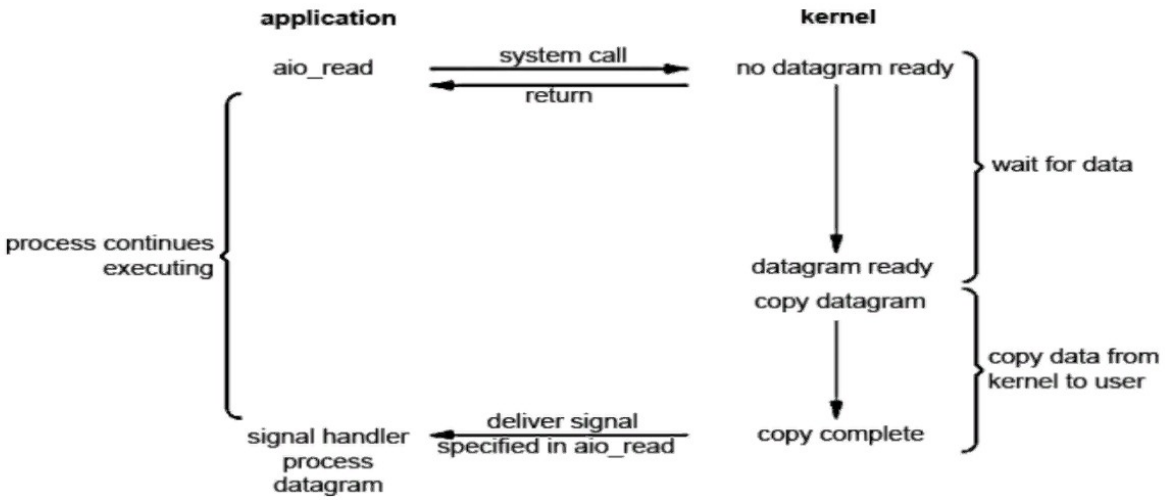
Результат выполнения системного вывода sigaction() возвращается сразу и приложение не блокируется. Всю работу берет на себя ядро:

- отслеживает готовность данных;
- затем посылает сигнал SIGIO, который вызывает установленный на него обработчик (функция обратного вызова – callback).

Сам вызов recvfrom() может быть выполнен либо в обработчике сигнала, который отправляет основному циклу информацию, что данные готовы, либо в основном потоке программы.

Сигнал SIGIO для каждого процесса может быть только один. В результате, за один раз можно обработать только один файловый дескриптор. Во время выполнения обработчика сигнала сигнал блокируется. Если за время блокировки сигнал доставляется несколько раз, то они теряются. Если маска сигнала sa_mask = NULL, то во время выполнения обработчика другие сигналы не блокируются.

5. Асинхронный ввод-вывод (asynchronous I/O: the POSIX aio functions)



Осуществляется с помощью специальных системных вызовов (функции реального времени), которые работают так, что сообщают ядру о начале операции и уведомляют приложение, когда вся операция, включая копирование данных из ядра в буфер приложения, завершена.

В вызове aio_read() даётся указание ядру начать операцию ввода-вывода, и указывается, каким сигналом уведомить процесс о завершении операции (включая копирование данных в пользовательский буфер). При этом вызывающий процесс не блокируется, а результат операции (например, полученная UDP-дейтаграмма) может быть обработан в обработчике сигнала.

Разница с предыдущей моделью ввода-вывода, который управляется сигналом, состоит в том, что в предыдущей модели сигнал уведомляет о возможности начала операции (вызове операции чтения), а в асинхронной модели сигнал уведомляет уже о завершении операции копирования данных в буфер пользователя.

Функции для асинхронного ВВ. У нас точно не было

Информация, необходимая для работы с асинхронным вводом-выводом, хранится в структуре aiocb (для 64-х битных операций существует аналогичная структура aiocb64):

```
struct aiocb { /* структура управления асинхронным вводом-выводом.
*/
     int aio_fildes; /* дескриптор файла */
     int aio_lio_opcode; /* выполняемая операция. */
     int aio_reqprio; /* смещение */
     volatile void *aio_buf; /* адрес буфера. */
     size_t aio_nbytes; /* длина передаваемой последовательности
*/
     struct sigevent aio_sigevent; /* номер и значение сигнала. */
```

```
    ...
}
```

Некоторые функции асинхронного ввода-вывода:

```
int aio_read(struct aiocb *__aiocbp);
int aio_write(struct aiocb *__aiocbp);
int lio_listio(int __mode, struct aiocb* const
list[__restrict_arr], int __nent, struct sigevent *__restrict
__sig);
int aio_cancel(int __fildes, struct aiocb *__aiocbp);
```

Параметры функций: файловый дескриптор, адрес буфера, номер сигнала sigev_signo и функция, используемая для уведомления потока. Функция lio_listio() позволяет инициализировать выполнение целой цепочки асинхронных операций (длиной __nent). Как и в случае с потоками, асинхронные операции проще запустить, чем позже остановить, поэтому для их остановки предлагается специальная функция aio_cancel(). Также можно предположить, что каждая асинхронная операция выполняется как отдельный поток.

В основном в асинхронном вводе-выводе внимание сосредотачивается на двух моментах:

1. на возможности определить, что ввод-вывод можно выполнить быстро;
2. на завершении операции ввода-вывода в любом случае: при невозможности выполнения ввода-вывода сразу возвращается ошибка.

Проблема асинхронного ввода-вывода состоит в том, что необходимо получать результаты асинхронных действий синхронно, т.е. асинхронные операции в итоге должны доставлять процессам данные, а процессы должны иметь возможность получать эти данные по необходимости.

Классификация моделей ввода-вывода

Операции ввода-вывода могут быть блокирующими и неблокирующими, синхронными и асинхронными.

	Блокирующие	Неблокирующие
Синхронные	read/write	**read/write (O_NONBLOCK)**
Асинхронные	I/O multiplexing (select/poll)	AIO

В соответствии с данной классификацией:

- блокирующий ввод-вывод – блокирующий синхронный;
- неблокирующий ввод-вывод или опрос (polling) – неблокирующий синхронный;
- мультиплексированный ввод-вывод – блокирующий асинхронный;
- асинхронный ввод-вывод – неблокирующий асинхронный.
- Ввод-вывод, управляемый сигналами, относится к неблокирующему асинхронному, но при получении сигнала о готовности данных вызывается блокирующий системный вызов.

Тут просто повторяется все то же, что и выше

Блокирующий синхронный ввод-вывод является наиболее распространённым типом ввода-вывода: при обращении процесса к внешнему устройству с помощью системного вызова ввода-вывода процесс блокируется в ожидании завершения операции ввода-вывода; для информирования процессора о завершении операции ввода-вывода контроллер внешнего устройства формирует сигнал, который по линии IRQ№ поступает на контроллер прерывания; контроллер формирует сигнал прерывания, который по линии шины данных поступает на выделенный pin процессора; в конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания и при его поступлении переходит на выполнение обработчика данного прерывания; обработчик прерывания должен сохранить данные, пришедшие от контроллера устройства, в буфере ядра, чтобы затем функции обратного вызова драйвера устройства доставили полученные данные в буфер приложения. Для этого приложение разблокируется.

В неблокирующем синхронном вводе-выводе приложение, выдавшее запрос ввода-вывода не блокируется, а делает многочисленные запросы, проверяя готовность данных.

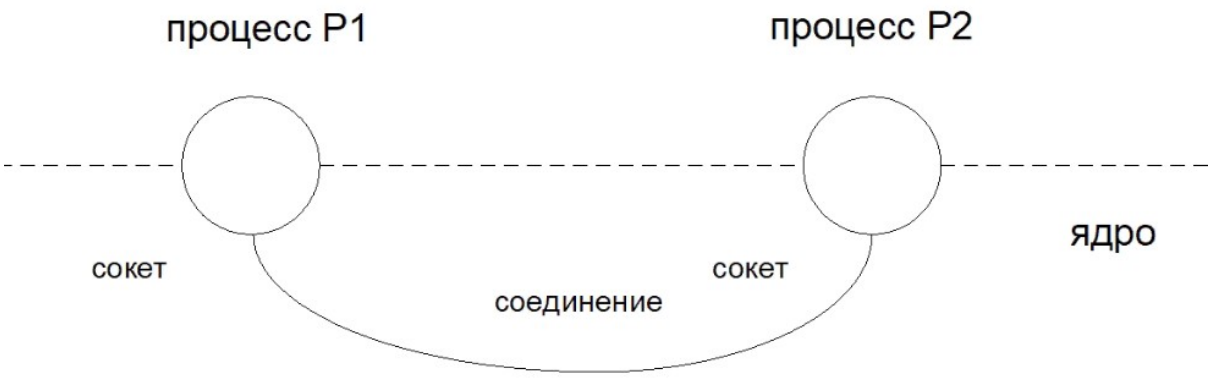
Другая парадигма блокировки – это неблокирующий ввод-вывод с блокировкой уведомления. В этой модели устанавливается неблокирующий ввод-вывод, а затем

вызывается блокирующий системный вызов мультиплексора (select или poll), чтобы определить какую-либо активность для файловых дескрипторов. Достоинством мультиплексора является то, что он используется для уведомления о многих дескрипторах.

Асинхронный ввод-вывод (AIO) - это метод выполнения операций ввода-вывода таким образом, что процесс, выдавший запрос ввода-вывода, не блокируется до завершения операции. Вместо этого после отправки запроса ввода-вывода процесс продолжает выполняться его код и можно позже проверить статус поданного запроса. Для завершения транзакции ввода-вывода может использоваться сигнал, на который устанавливается обработчик сигнала, или поток на основе обратного вызова.

Средства взаимодействия процессов — сокеты Беркли.

Сокет – абстракция конечной точки соединения (коммуникации, взаимодействия).



Абстракция сокетов была введена в BSD Unix (Berkley Software Distribution), поэтому сокеты часто называют сокетами BSD или Berkley.

Сокеты используются для взаимодействия параллельных процессов по модели клиент-сервер как на одной машине машине, так и в распределенных системах (сетевые сокеты). Интерфейс единообразен.

Мы рассматриваем 2 вида: сокеты в пространстве имен и сетевые сокеты (есть еще парные, как альтернатива pipe. Программные каналы pipe – базовое средство передачи информации между процессами).

Создание сокета — семейство, тип, протокол.

```
#include <sys/types.h>
#include <sys/socket.h>
sockfd = socket(int socket_family, int socket_type, int protocol);
```

Системный вызов socket() создаёт так называемую конечную точку соединения и возвращает файловый дескриптор (int), который относится к этой конечной точке. При успешном вызове возвращается файловый дескриптор с наименьшим номером дескриптора файла, который в данный момент еще не открыт для процесса.

offtop: возвращает файловый дескриптор, потому что парадигма все – файл: отобразить все объекты, к которым обращаются для чтения и записи, в виде файла. Чтобы использовать всего 2 функции – read и write.

Параметры:

1.domain или family

Задает домен соединения или семейство адресов (address family - AF). Домен определяет семейство протоколов, которое будет использоваться для связи. Семейства описаны в <sys/socket.h>. В настоящее время определены такие форматы:

- AF_UNIX – семейство юникс. Сокеты для межпроцессного взаимодействия на отдельно стоящей (локальной) машине, создаются в файловом пространстве имен, то есть можем увидеть в файловой подсистеме (обозначаются s). Входят в 7 типов файлов, поддерживаемых в юникс.
- AF_INET – сетевые сокеты по протоколу TCP/IP (любая IP сеть) для сетей IPv4) (также называется интернет домен)
- AF_INET6 - сетевые сокеты по протоколу TCP/IP для сетей IPv6.
- AF_IPX - IPX for local area network environments
- AF_UNSPEC – неопределенный домен (неопределенное семейство адресов)
- и некоторые другие.

(табличка из методы с остальными семействами)

Название	*Назначение*
AF_UNIX, AF_LOCAL	*Локальное соединение*
AF_INET	*Протоколы Интернет IPv4*
AF_INET6	*Протоколы Интернет IPv6*
AF_IPX	*Протоколы Novell IPX*
AF_NETLINK	*Устройство для взаимодействия с ядром*
AF_X25	*Протокол ITU-T X.25/ISO-8208*
AF_AX25	*Протокол любительского радио AX.25*
AF_ATMPVC	*Доступ к низкоуровневым PVC в ATM*
AF_APPLETALK	*AppleTalk*
AF_PACKET	*Низкоуровневый пакетный интерфейс*
AF_ALG	*Интерфейс к ядерному крипто-API*

2.тип сокета (type)

Определяет тип нужного коммуникационного отношения (семантику соединения) (в конце файла описано подробней). В настоящее время определены следующие 6 типов:

1. SOCK_STREAM - Обеспечивает создание надежного упорядоченного полнодуплексного логического соединения между двумя сокетами. Может также поддерживаться механизм внепоточных данных.

SPECIAL FOR RYAZANOVA (https://www.opennet.ru/man.shtml?topic=socket&category=2&russian=0): Сокеты типа SOCK_STREAM являются соединениями полнодуплексных байтовых потоков, похожими на каналы. Они не сохраняют границы записей. Потоковый сокет должен быть в состоянии соединения перед тем, как из него можно будет отсылать данные или принимать их в нем. Соединение с другим сокетом создается с помощью системного вызова connect(2).

2. SOCK_DGRAM – определяют ненадежную службу datagram без установления логического соединения, когда пакеты могут передаваться без сохранения логического порядка – широковещательная передача данных. Ограниченная длина сообщения

3. SOCK_SEQPACKET - Обеспечивает работу последовательного двустороннего канала для передачи дейтаграмм на основе соединений; дейтаграммы имеют постоянный размер; от получателя требуется за один раз прочитать целый пакет.

4. SOCK_RAW - Обеспечивает прямой доступ к сетевому протоколу.
   Низкоуровневый интерфейс datagram по протоколу IP и не обязательно POSIX1

5. SOCK_RDM - Обеспечивает надежную доставку дейтаграмм без гарантии, что они будут расположены по порядку.

6. SOCK_PACKET - Этот тип устарел и не должен использоваться в новых программах;

Некоторые типы сокетов могут быть не реализованы во всех семействах протоколов. Начиная с Linux 2.6.27, аргумент type предназначается для двух вещей: кроме определения типа сокета, для изменения поведения socket() он может содержать побитовую сумму любых следующих значений (как флаги open):

- SOCK_NONBLOCK - устанавливает флаг состояния файла O_NONBLOCK для нового открытого файлового дескриптора. Использование данного флага заменяет дополнительные вызовы fcntl(2) для достижения того же результата.

(про флаг в контексте open) Если O_NONBLOCK установлен, open возвращается немедленно. O_NONBLOCK бит также воздействует на чтение и на запись: он разрешает им возвращаться немедленно с состоянием ошибки, если не имеется никакого доступного ввода, или если вывод не может быть записан.

- SOCK_CLOEXEC - устанавливает флаг close-on-exec (FD_CLOEXEC) для нового открытого файлового дескриптора.

(про флаг в контексте open) Если бит FD_CLOEXEC установлен в 0, то файл будет оставлен открытым при вызове exec, в противном случае он будет закрыт.

(шо такое эта ваша дуплексность). При полудуплексной связи передача данных в каждый момент времени возможна только в одном направлении (т. е. их можно либо передавать, либо принимать), тогда как при полнодуплексной связи данные можно передавать и принимать одновременно.

3.Protocol

Задаётся определённый протокол, используемый с сокетом. Обычно, только единственный протокол существует для поддержки определенного типа сокета с заданным семейством протоколов.
(Например, для SOCK_STREAM это TCP (transmission control protocol), для SOCK_DGRAM - UDP (user datagram protocol))

Обычно для этого параметра устанавливается значение 0, в этом случае протокол устанавливается по умолчанию для соответствующего семейства и типа. Однако, может существовать несколько протоколов.

Ошибки

В случае успешного выполнения возвращается дескриптор, ссылающийся на сокет. В случае ошибки возвращается -1, а значение errno устанавливается соответствующим образом.

ОШИБКИ
EACCES Нет прав на создание сокета указанного типа и/или протокола
EAFNOSUPPORT Реализация не поддерживает указанное семейства адресов.
EINVAL Неизвестный протокол или недоступное семейство протоколов.
EINVAL Неверные флаги в type.
EMFILE Было достигнуто ограничение по количеству открытых файловых дескрипторов на процесс.
ENFILE Достигнуто максимальное количество открытых файлов в системе.
ENOBUFS или ENOMEM Недостаточно памяти для создания сокета. Сокет не может быть создан, пока не будет освобождено достаточное количество ресурсов.
EPROTONOSUPPORT Тип протокола или указанный протокол не поддерживаются в этом домене.

sys_socketcall

Ядро Linux предоставляет единственный системный вызов:

```
<net/socket.h>
```

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
{
      unsigned long a[6];
      unsigned long a0,a1;
      int err;

      if(call<1||call>SYS_RECVMSG)
                return -EINVAL;

      /* copy_from_user should be SMP safe. */
      if (copy_from_user(a, args, nargs[call])) //это подчеркнуть
                return -EFAULT;

      a0=a[0];
      a1=a[1];

      switch(call)
      {
         //похоже на sys_define3 (это комментарий X)
         case SYS_SOCKET:
              err = sys_socket(a0,a1, a[2]);
              break;
         case SYS_BIND:
              err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
             break;
         case SYS_CONNECT:
              err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
         ...
         default:
               err = -EINVAL;
               break;
      }
      return err;
}
```

Дизайн сокетов Беркли следует парадигме UNIX: в идеале отобразить все объекты, к которым осуществляется доступ для чтения или записи, на файлы, чтобы с ними можно было работать с использованием обычных функций записи и чтения в/из файла.

Пытаясь понять, что вообще происходит, нашла эту функцию с комментарием: socketcall() is a common kernel entry point for the socket system calls.

```
#include <linux/net.h>
int socketcall(int call, unsigned long *args);
```

socketcall – это видимо оболочка для sys_socketcall, и тип любая функция на сокете (socket(), bind(), connect(), ..) на самом деле выполняется через sys_socketcall

Системный вызов sys_socket()

Функция sys_socket() создает сокет. Данная функция инициализирует структуру socket. Значение retval и есть тот самый дескриптор, который возвращает функция socket().

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
	int retval;
	struct socket *sock;

	retval = sock_create(family, type, protocol, &sock);
	if (retval < 0)
		goto out;

	retval = sock_map_fd(sock);
	if (retval < 0)
		goto out_release;

out:
	/* It may be already another descriptor 8) Not kernel
problem. */
	return retval;

out_release:
	sock_release(sock);
	return retval;
}
```

(PS. обожаю эти комментарии разработчиков, особенно со смайликами))

struct socket – структура, описывающая сокет

```
/**
*  struct socket - general BSD socket
*  @state: socket state (%SS_CONNECTED, etc)
*  @type: socket type (%SOCK_STREAM, etc)
*  @flags: socket flags (%SOCK_NOSPACE, etc)
*  @ops: protocol specific socket operations
*  @file: File back pointer for gc
*  @sk: internal networking protocol agnostic socket
representation
*  @wq: wait queue for several uses
*/
struct socket {
	socket_state		state; //5 состояний (ниже
рассматриваются)
```

```
      short                   type; //тип сокета - 2 параметр функции
socket

      unsigned long           flags; //используется для синхронизации
доступа (те самые флаги O_CLOEXEC и O_NONBLOCK)
      struct socket_wq __rcu  *wq;

      struct file       *file; //указатель на дескриптор открытого
файла (парадигма все файл: сокет - специальный файл), а file имеет
указатель на struct inode
      struct sock             *sk;
      const struct proto_ops  *ops; /*операции, связанные с
протоколом (так называемый сетевой протокол) (например, TCP
(transmission control) или UDP (user datagram)). Структура подобна
struct file_operations - функции, определенные на сокете. Зависят
от типа и протокола*/
};
```

Состояния сокета.

Поле (структуры socket) socket_state state определяет одно из 5 возможных состояний:

1. SS_FREE – не занят
2. SS_UNCONNECTED – не соединен
3. SS_CONNECTING – соединяется в данный момент
4. SS_CONNECTED – соединен
5. SS_DISCONNECTING – разъединяется в данный момент

Адресация сокетов и ее особенности для разных типов сокетов.

Форматы адресов

Каждый сокетный домен имеет свой формат сокетных адресов, выраженный в отдельной адресной структуре.
Каждая из этих структур начинается с целочисленного поля «семейства» (с типом sa_family_t), в котором указывается тип адресной структуры. Это позволяет различным системным вызовам определить домен конкретного сокетного адреса.

- Для передачи сокетного адреса любого типа через программный интерфейс сокетов служит тип struct sockaddr. Структура определяет адрес в самом общем виде.

```
struct sockaddr
{
      unsigned short sa_family; // Семейство адресов, AF_xxx.
Определяет семейство адресов, но точный формат не определен.
```

```
      char sa_data[14]; // 14 байтов для хранения адреса. Вид
адреса зависит от выбранного вами домена.
  };
```

Целью данного типа является приведение типов сокетных адресов определенного домена к «общему» типу, что позволяет избежать предупреждений компилятора о несовпадении типов в вызовах API сокетов.

(В switch в разных функциях используется указатель struct sockaddr)

В зависимости от назначения программы вместо sockaddr используют: sockaddr_in и sockaddr_un (internet и unix соответственно):

- sockaddr_un для домена unix. Адрес в этом домене - это текстовая строка - имя файла, через который происходит обмен данными.

```
struct sockaddr_un
{
    sa_family_t sun_family;
    char sun_path[ 108 ];
}
```

- Для сетевого взаимодействия (взаимодействия в интернете) определена структура sockaddr_in. В Internet-домене адрес задается комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

```
struct sockaddr_in
 {
      short int sin_family; // Семейство адресов (AF_INET).
Соответствует полю sa_family в sock_addr
      unsigned short int sin_port; // Номер порта. ПОРТ-ЭТО АДРЕС
      struct in_addr sin_addr; // IP-адрес хоста (адрес в инете,
который пишется через . (является структурой, ниже)
    unsigned char sin_zero[8]; // Дополнение до размера структуры
sockaddr.  почему именно 8: sizeof(struct sockaddr)-sizeof(sa_family_t) –
sizeof(uint16_t)-sizeof(struct in_addr)
 };
```

```
struct in_addr {
      unsigned long s_addr; // по мнению X тип _u32
};
```

Зачем понадобилось заключать всего одно поле в структуру? Дело в том, что раньше in_addr представляла собой объединение (union), содержащее гораздо большее число полей. Сейчас, когда в ней осталось всего одно поле, она продолжает использоваться для обратной совместимости.

Для задания адреса можно воспользоваться одной из следующих констант:

- INADDR_ANY — все адреса локального хоста (0.0.0.0);Благодаря этой константе наша программа сервер зарегистрируется на всех адресах той машины, на которой она выполняется.
- *INADDR_LOOPBACK — адрес loopback интерфейса (127.0.0.1);*
- *INADDR_BROADCAST — широковещательный адрес (255.255.255.255).*

Связь сокета с адресом

Функция socket() создает "безымянный" сокет, т.е. не связанный ни с локальным адресом, ни с номером порта. Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру называют именованием сокета). Иногда связывание осуществляется неявно (внутри функций connect и accept), но выполнять его необходимо во всех случаях.

Фактически нужно идентифицировать процесс, в котором выполняется соединение для процесса. Идентификация состоит из 2 частей.

1. Идентификация сетевого узла с помощью сетевого адреса.
2. Идентификация конкретного процесса с помощью номера службы.

Для явного связывания сокета с некоторым адресом используется функция bind. Ее прототип имеет вид:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

В качестве первого параметра передается дескриптор сокета, который привязывается к заданному адресу. Второй параметр – addr - содержит указатель на структуру struct sockaddr, а третий - длину этой структуры.

Аппаратный и сетевой порядок байтов.

Существует два порядка хранения байтов в слове (2 байта) и двойном слове (4 байта)

- порядок хоста (host byte order) (little-endian),
- сетевой порядок (network byte order) хранения байтов (big-endian).

4-байтное целое число 0x01020304 будет сохранено в памяти системы big endian следующим образом:

Байт0	Байт1	Байт2	Байт3
0x01	0x02	0x03	0x04

в памяти системы little endian:

Байт0	Байт1	Байт2	Байт3
0x04	0x03	0x02	0x01

Обычно система сама все решает, но когда нужно обмениваться данными с другой системой, обе системы должны учитывать формат хранения данных в памяти (endianness).

Linux kernel может быть либо big endian, либо little endian, в зависимости от архитектуры, в расчете на которую kernel скомпилировано.

При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой. Для этого используются функции

- htons (Host To Network Short, 16 разрядов) и
- htonl (Host To Network Long, 32).

Обратное преобразование выполняют функции

- ntohs (Network To Host Short) и
- ntohl.

(если endianness совпадают, то просто не будут выполняться)

Пример из лабы

(что именно нужно этими функциями преобразовывать, ибо мне было непонятно)
сервер

```
#define PORT 3425

int main(void)
{
     int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
     if (sock_fd == -1)
     {
       printf("socket() failed\n");
       return EXIT_FAILURE;
     }

     struct sockaddr_in serv_addr;
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY; //сервер
зарегистрируется на всех адресах той машины, на которой она
выполняется.
     serv_addr.sin_port = htons(PORT);
```

```
     if (bind(sock_fd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) == -1)
     {
       printf("bind() failed\n");
        del_socket();
       return EXIT_FAILURE;
     }
…
```

клиент

В программе-клиенте, прежде всего, нужно решить задачу, с которой не было при написании сервера, а именно выполнить преобразование доменного имени сервера в его сетевой адрес. Разрешение доменных имен выполняет функция gethostbyname():

```
#define PORT 3425
#define SOCK_ADDR "localhost"

int main(void)
{
    int sock_fd;
    struct sockaddr_in serv_addr;
    struct hostent *host;

    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd < 0)
    {
      printf("socket() failed\n");
      return EXIT_FAILURE;
    }

     //преобразование доменного имени сервера в его сетевой
адрес.
    host = gethostbyname(SOCK_ADDR);
    if (!host)
    {
        printf("gethostbyname() failed\n");
        return EXIT_FAILURE;
    }

    serv_addr.sin_family = AF_INET; // Семейство адресов
    //главное отличие sockaddr_in от sockaddr_un - наличие
параметра sin_port, предназначенного для хранения значения
```

```
порта.
    serv_addr.sin_port = htons(PORT); // Номер порта
    serv_addr.sin_addr = *((struct in_addr*) host-
>h_addr_list[0]); // IP-адрес

    // Для установления активного соединения по переданному
адресу.
     if (connect(sock_fd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) == -1)
    {
      printf("connect() failed\n");
      return EXIT_FAILURE;
    }
…
```

Модель клиент-сервер.

В модели клиент-сервер роли определены: сервер предоставляет ресурсы и службы одному или нескольким клиентам, который обращаются к серверу за обслуживанием.

Когда клиент запрашивает соединение с сервером, сервер может либо принять, либо отклонить это соединение. Если соединение принято, то сервер устанавливает и поддерживает соединение с клиентом по определенному протоколу.

Клиент не предоставляет общий доступ ни к одному из своих ресурсов, но запрашивает данные или службу у сервера.

Взаимодействие процессов через сокеты выполняется как раз по модели клиент-сервер.

Роли сервера и клиентов различаются с точки зрения поддержки коммуникационных отношений: клиент активно устанавливает соединения с сервером, сервер сначала пассивно ожидает поступления входящих запросов на установку соединения, а при поступлении запроса сервер его фиксирует и начинает обрабатывать; обработав запрос, сервер посылает ответ клиенту.

Таким образом, на стороне клиента и на стороне сервера в интерфейсе сокетов выполняется разная последовательность системных вызовов. (что мы увидим на сетевом стеке)

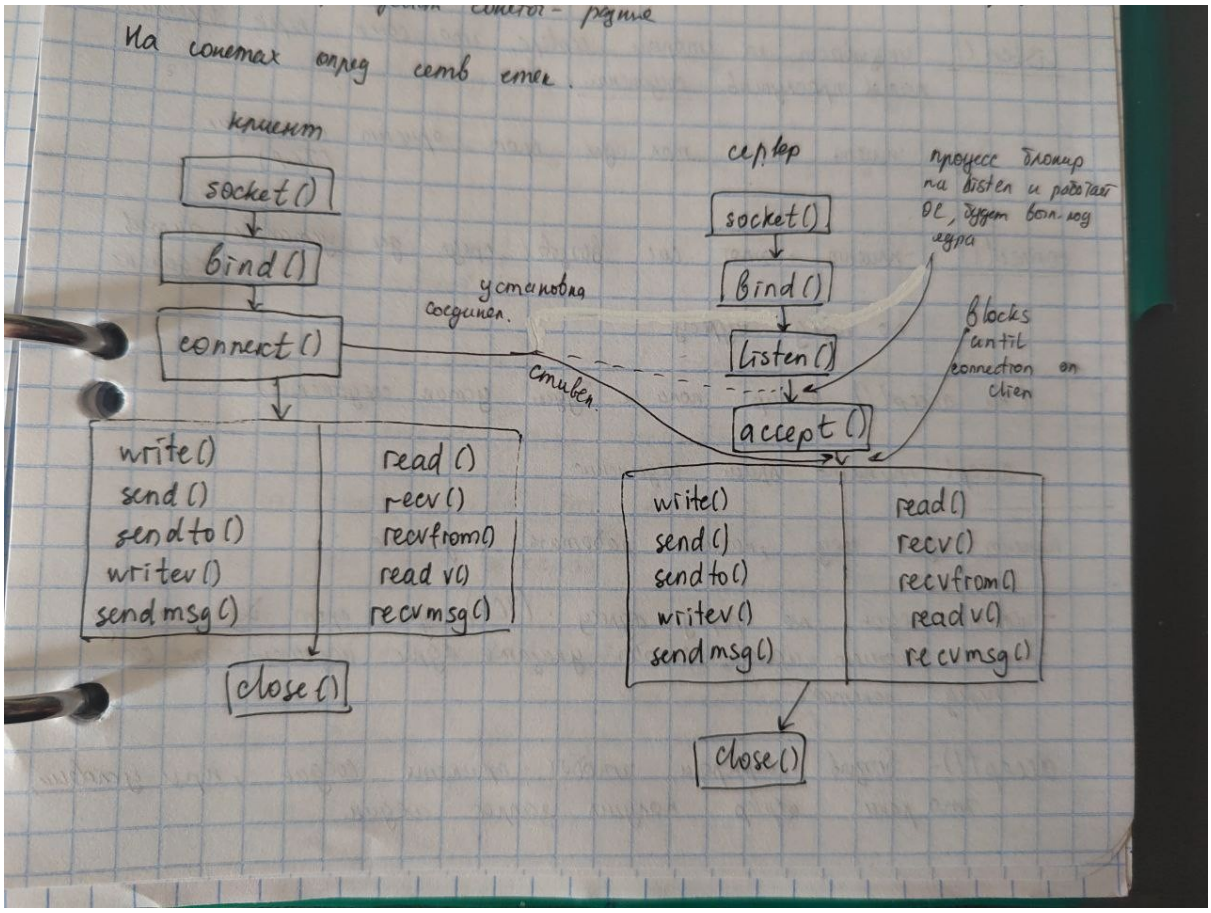
Сетевые сокеты — сетевой стек.

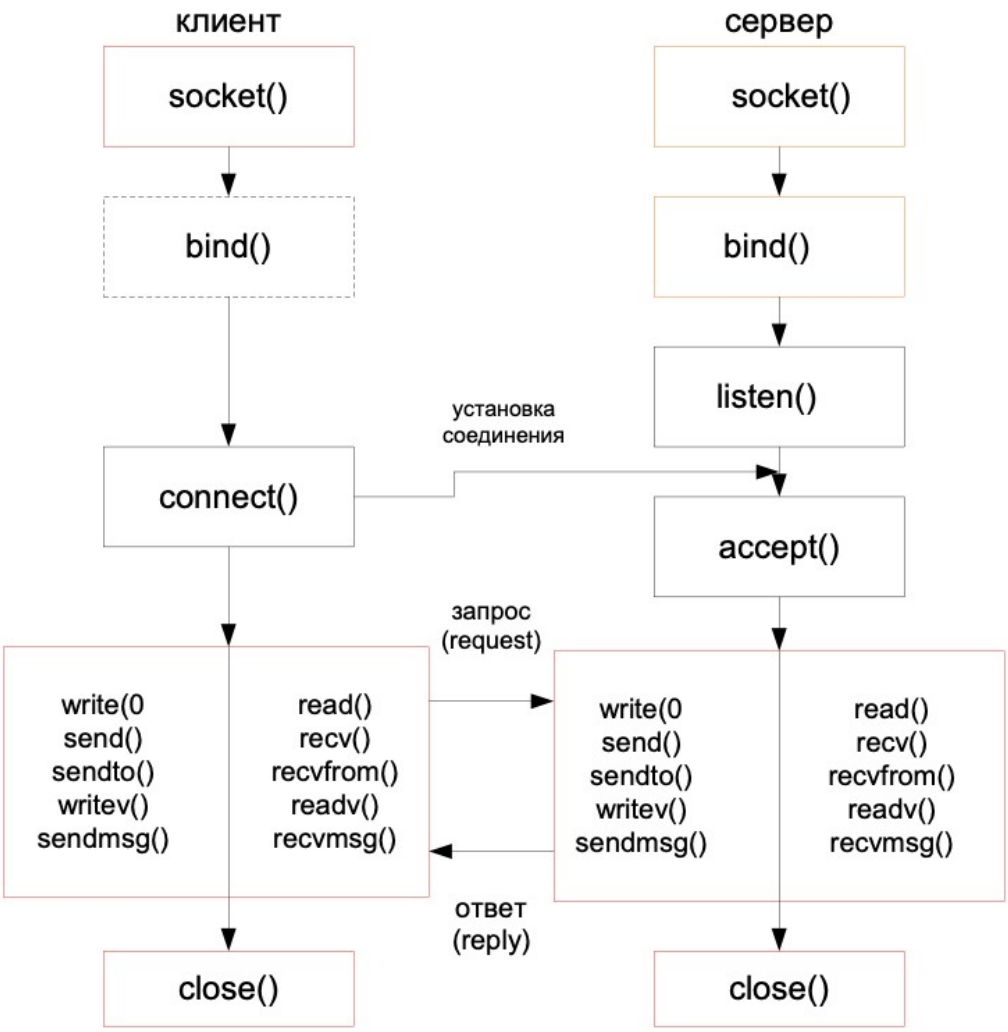
Сетевой стек

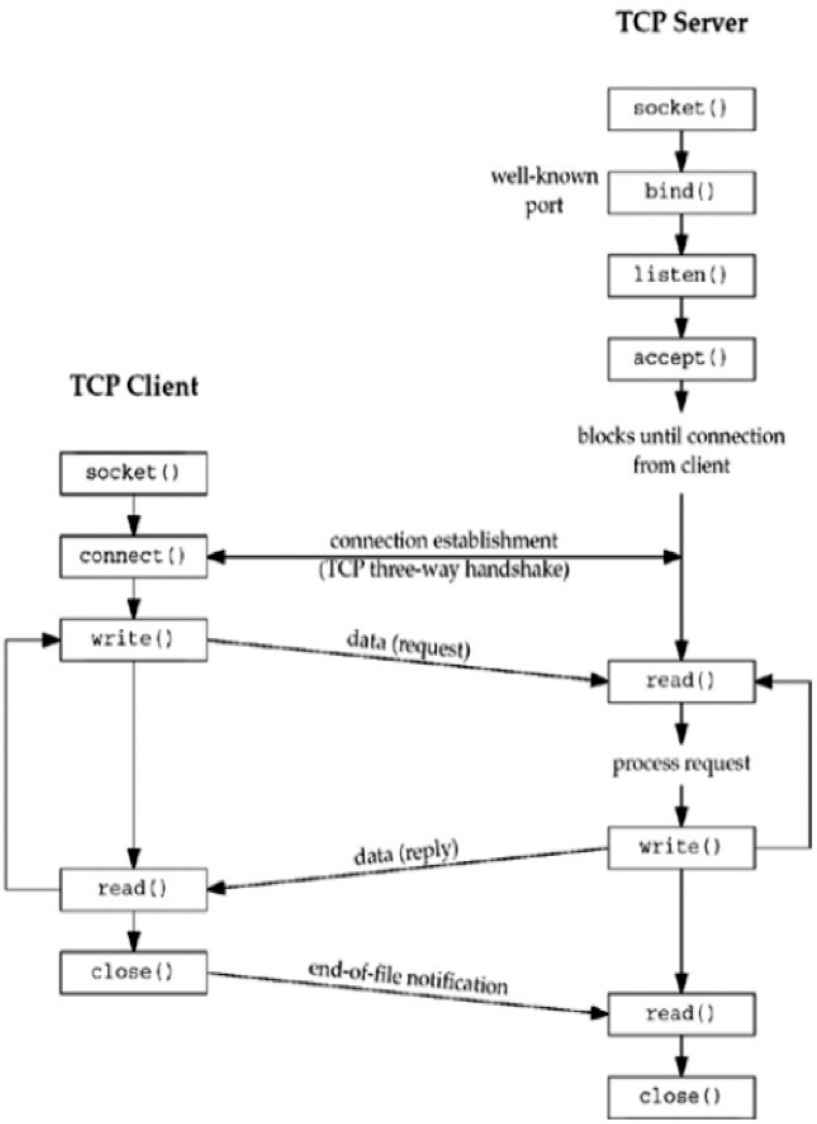
Картинки про одно и то же, немного разные стрелочки, вот комментарий по этому поводу.

Стрелка с подписью “установка соединения”: куда ее вводить справа: просто стрелкой ПОСЛЕ accept, а потом пунктиром ДО.

Что значит после: blocks until connection on client. То есть процесс блокируется (состояние пассивного ожидания) на listen, а работает ОС (код ядра). Когда приходит соединение, accept принимает УСТАНОВИВШЕЕСЯ соединение




клиент
сервер
socket()
socket()
bind()
bind()
listen()
установка
соединения
connect()
accept()
запрос
(request)
write(0
send()
sendto()
writev()
sendmsg()
read()
recv()
recvfrom()
readv()
recvmsg()
write(0
send()
sendto()
writev()
sendmsg()
read()
recv()
recvfrom()
readv()
recvmsg()
ответ
(reply)
close()
close()



close еще раз напоминает – все файл (даже сетевой сокет)

сюда же можно: В целом: функции, определенные на сокетах

сетевые сокеты

(не думаю, что этот пункт вообще надо писать вот так отдельно, но все же)

Сетевые сокеты являются универсальным типом сокетов. Система приложений, предназначенных для работы даже на одном компьютере, может использовать сетевые сокеты для обмена данными.

В качестве примера рассматривается комплекс из двух приложений, клиента и сервера, использующих сетевые сокеты для обмена данными.

Сервер

Прежде всего, надо получить дескриптор сокета:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
if (socket < 0) {
    printf("socket() failed: %d\n", errno);
    return EXIT_FAILURE;
}
```

Затем вызывается функция bind():

```
serv_addr.sin_family = AF_INET; //serv_addr - это struct
sockaddr_in
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);
if (bind(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
< 0) {
    printf("bind() failed: %d\n", errno);
    return EXIT_FAILURE;
}
```

Сетевой сервер должен уметь обрабатывать запросы множества клиентов одновременно. При этом в соединениях «точка-точка», например, при использовании потоковых сокетов, для каждого клиента у сервера должен быть открыт отдельный сокет. Для обеспечения такой возможности вызывается функция listen(2), которая переводит сервер в режим ожидания запроса на соединение:

```
listen(sock, 1);
```

Второй параметр listen() – максимальное число соединений, которые сервер может обрабатывать одновременно.

Затем вызывается функция accept(2), которая устанавливает соединение в ответ на запрос клиента и создает копию сокета для того, чтобы исходный сокет мог продолжать прослушивание:

```
newsock = accept(sock, (struct sockaddr *) &cli_addr, &clen);
if (newsock < 0)
{
   printf("accept() failed: %d\n", errno);
   return EXIT_FAILURE;
}
```

Получив запрос на соединение, функция accept() возвращает новый сокет, открытый для обмена данными с клиентом, запросившим соединение. Сервер как бы перенаправляет запрошенное соединение на другой сокет, оставляя сокет sock свободным для прослушивания запросов на установку соединения. Второй параметр функции accept() содержит сведения об адресе клиента, запросившего соединение, а третий параметр указывает размер второго. Так же как и при вызове функции recvfom(), может быть передано значение NULL в последнем и предпоследнем параметрах. Для чтения и записи данных сервер использует функции read() и write(), а для закрытия сокетов, естественно, close().

Клиент

В программе-клиенте, прежде всего, нужно решить задачу, с которой не было при написании сервера, а именно выполнить преобразование доменного имени сервера в его сетевой адрес. Разрешение доменных имен выполняет функция gethostbyname():

```
server = gethostbyname(argv[1]);
if (server == NULL)
 {
    printf("Host not found\n");
    return EXIT_FAILURE;
 }
```

Функция получает указатель на строку с Интернет-именем сервера и возвращает указатель на структуру hostent (переменная server), которая содержит имя сервера в приемлемом для дальнейшего использования виде. При этом, если необходимо, выполняется разрешение доменного имени в сетевой адрес.

Далее заполняются поля переменной serv_addr (структуры sockaddr_in) значениями адреса и порта:

```
serv_addr.sin_family = AF_INET;
strncpy((char *)&serv_addr.sin_addr.s_addr,
(char *)server->h_addr, server->h_length);
serv_addr.sin_port = htons(port);
```

Программа-клиент открывает новый сокет с помощью вызова функции socket() и вызывается функция connect(2) для установки соединения:

```
if (connect(sock, &serv_addr, sizeof(serv_addr)) < 0)
 {
    printf("connect() failed: %d", errno);
    return EXIT_FAILURE;
 }
```

Теперь сокет готов к передаче и приему данных.

По умолчанию функция socket() создает блокирующий сокет. Чтобы сделать его не- блокирующим, надо использовать функцию fcntl(2) с флагом O_NONBLOCK:

```
sock = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sock, F_SETFL, O_NONBLOCK);
```

Теперь любой вызов функции read() для сокета sock будет возвращать управление сразу же. Если на входе сокета нет данных для чтения, функция read() вернет значение EAGAIN.

Сокеты AF_UNIX =Сокеты в файловом пространстве имен

Сокеты в файловом пространстве имен – супер название, отражающее суть. Остается добавить, что взаимодействуют через специальные файлы (файловую подсистему).

Это сокеты для межпроцессного взаимодействия на отдельно стоящей (локальной) машине. Они используют в качестве адресов имена файлов (специального типа сокет) в файловом пространстве имен, то есть можем увидеть в файловой подсистеме (обозначаются s). Входят в 7 типов файлов, поддерживаемых в юникс. Также они могут быть и безымянными

Допустимые типы сокета для домена UNIX:

- потоковый сокет SOCK_STREAM,
- датаграмный сокет SOCK_DGRAM, сохраняющий границы сообщений (в большинстве реализаций UNIX, доменные датаграмные сокеты UNIX всегда надёжны и не меняют порядок датаграмм);
- (начиная с Linux 2.6.4) ориентированный на соединение задающий последовательность пакетам сокет SOCK_SEQPACKET, сохраняющий границы сообщений и доставляющий сообщения в том же порядке, в каком они были отправлены.

Примеры

В следующем фрагменте кода создаем сокет и связываем его с файлом socket.soc.

```
//Создадим сокет (файловый дескриптор)
sock_fd = sock(AF_UNIX, SOCK_DGRAM, 0);
if (sock_fd == -1)
{
        perror("failed");
        return EXIT_FAILURE;
}

//заполнение структуры sockaddr
s_name.sa_family = AF_UNIX; //семейство
strcpy(s_name.sa_data, "socket.soc"); //имя специального
файла типа сокет

//установка связи с адресом (связываем сокет с файлом). То
есть взаимодействие по модели клиент-сервер для данного
семейства выполняется через пространство  файловых  имен
if (bind(sock_fd, &s_name, strlen(s_name.sa_data) +
sizeof(s_name.sa_family)) < 0)
{
        perror("bind failed");
        return EXIT_FAILURE;
}
```

```
//После вызова bind() программа-сервер становится доступна для соединения по заданному адресу (имени файла).
```

При обмене данными используются специальные функции recvfrom(2) и sendto(2). Эти же функции могут применяться и при работе с потоковыми сокетами.

По завершении работы с сокетом он «закрывается» с помощью «файловой» функции close(). Перед выходом из программы-сервера следует удалить файл сокета, созданный в результате вызова socket(), что делается с помощью функции unlink().

Для чтения данных из датаграммного сокета используется функция recvfrom(2), которая по умолчанию блокирует программу до тех пор, пока на входе не появятся новые данные. При вызове функции recvfrom() ей передается указатель на еще одну структуру типа sockaddr, в которой функция возвращает данные об адресе клиента, запросившего соединение (в случае файловых сокетов этот параметр не несет полезной информации). Последний параметр функции recvfrom() – указатель на переменную, в которой будет возвращена длина структуры с адресом. Если информация об адресе клиента не нужна, то можно передать значения NULL в предпоследнем и последнем параметрах.

```
bytes = recvfrom(sock, buf, sizeof(buf), 0, &rcvr_name, &namelen);
```

В программе-клиента (fsclient.c) открывается сокет с помощью функции socket() и передаются данные (текстовую строку) серверу с помощью функции sendto(2):

Первый параметр функции sendto() – дескриптор сокета, второй и третий параметры позволяют указать адрес буфера для передачи данных и его длину. Четвертый параметр предназначен для передачи дополнительных флагов. Предпоследний и последний параметры несут информацию об адресе сервера и его длине, соответственно.

Если при работе с датаграммными сокетами вызвать функцию connect(2) (см. ниже), то можно не указывать адрес назначения каждый раз (достаточно указать его один раз, как параметр функции connect()). Перед вызовом функции sendto() нам надо заполнить структуру sockaddr (переменную srvr_name) данными об адресе сервера. После окончания передачи данных сокет закрывается с помощью close(). Если запустить программу-сервер, а затем программу-клиент, то сервер распечатает тестовою строку, переданную клиентом.

```
srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, SOCK_NAME);
strcpy(buf, "Hello, Unix sockets!");
sendto(sock, buf, strlen(buf), 0, &srvr_name,
strlen(srvr_name.sa_data) + sizeof(srvr_name.sa_family));
```

Мультиплексирование

(сначала по-хорошему описать общую идею мультиплексирования)

Существуют 2 способа, как организовать параллельное обслуживание сервером нескольких клиентов *(что собственно и есть мультиплексирование)*

Способ 1

(я его привожу, так как X упоминала на семинаре, назвав “листинг 6 у шаргина)

Этот способ подразумевает создание дочернего процесса для обслуживания каждого нового клиента. При этом родительский процесс занимается только прослушиванием порта и приёмом соединений.

Чтобы добиться такого поведения,

- сразу после accept
- сервер вызывает функцию fork для создания дочернего процесса.
- Далее анализируется значение, которое вернула эта функция. В родительском процессе оно содержит идентификатор дочернего, а в дочернем процессе равно нулю. Используя этот признак, мы
    - переходим к очередному вызову accept в родительском процессе,
    - а дочерний процесс обслуживает клиента и завершается (_exit).

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int sock, listener;
    struct sockaddr_in addr;
    char buf[1024];
    int bytes_read;

    listener = socket(AF_INET, SOCK_STREAM, 0);
    if(listener < 0)
    {
        perror("socket");
        exit(1);
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(3425);
    addr.sin_addr.s_addr = INADDR_ANY;
```

```
    if(bind(listener, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("bind");
        exit(2);
    }

    listen(listener, 1);

    while(1)
    {
        sock = accept(listener, NULL, NULL);
        if(sock < 0)
        {
            perror("accept");
            exit(3);
        }


        switch(fork())
        {

        case -1:
            perror("fork");
            break;

        // дочерний процесс
        case 0:
            close(listener);
            while(1)
            {
                bytes_read = recv(sock, buf, 1024, 0);
                if(bytes_read <= 0) break;
                send(sock, buf, bytes_read, 0);
            }

           close(sock);
           _exit(0);

        // родительский процесс - переходим к очередному вызову
accept
        default:
            close(sock);
        }
    }
    close(listener);
```

```
    return 0;
}
```

Достоинства:

- компактные, понятные программы, в которых код установки соединения отделен от кода обслуживания клиента.

Недостатки

- если клиентов очень много, создание нового процесса для обслуживания каждого из них может оказаться слишком дорогостоящей операцией.
- такой способ неявно подразумевает, что все клиенты обслуживаются независимо друг от друга. Однако это может быть не так.
  Если, к примеру, вы пишете чат-сервер, то ваша основная задача - поддерживать взаимодействие всех клиентов, присоединившихся к нему. В этих условиях границы между процессами станут для вас серьезной помехой.

Способ 1_2

От Х. Видимо, это тот же способ, только вместо процессов можно использовать потоки

Наиболее часто для мультиплексирование на стороне сервера запускаются потоки. Но это неэффективно – они тяжеловесные: все создается clone (то есть речь идет о наследовании) (создают ли таблицы страниц собственные для потоков, как для процессов??? По идее - нет). Часто используют light weight processes (LWP) - создать поток и передать его потоку ядра. Есть pthread_create.

Способ 2

Второй способ основан на использовании неблокирующих сокетов (nonblocking sockets) и функции select.

Преимущество мультиплексирования – мы не ожидаем конкретный сокет, а выбираем первый готовый (не ждем)

(по методе сокет надо сначала сделать неблокирующимся. Но ты мне передала слова Х: нет смысла делать сокеты не блокирующимися, так как мультиплексор итак не блокирующийся)

Что такое неблокирующие сокеты

Сокеты, которые мы до сих пор использовали, являлись блокирующими (blocking): на время выполнения операции с таким сокетом ваша программа блокируется.

Например, если вы вызвали recv, а данных на вашем конце соединения нет, то в ожидании их прихода ваша программа "засыпает". Аналогичная ситуация

наблюдается, когда вы вызываете accept, а очередь запросов на соединение пуста. Это поведение можно изменить, используя функцию fcntl.

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

Эта несложная операция превращает сокет в неблокирующий.

Вызов любой функции с таким сокетом будет возвращать управление немедленно. Причём если затребованная операция не была выполнена до конца, функция вернёт -1 и запишет в errno значение EWOULDBLOCK.

Если на входе сокета нет данных для чтения, функция read() вернет значение EAGAIN.

Способ 2_1: polling

Чтобы дождаться завершения операции, мы можем опрашивать все наши сокеты в цикле, пока какая-то функция не вернёт значение, отличное от EWOULDBLOCK. Как только это произойдет, мы можем запустить на выполнение следующую операцию с этим сокетом и вернуться к нашему опрашиваемому циклу. Такая тактика (называемая в англоязычной литературе polling) работоспособна, но очень неэффективна, поскольку процессорное время тратится впустую на многократные (и безрезультатные) опросы.

(что было выше о polling – из Шаргина. Думаю, это стоит писать. В 2_3 poll и epoll – из интернета. Нам вообще ничего не говорили и не отправляли об этом. Писать или нет весь подпункт 2_3 – на выбор. Можно коротенечко упомянуть, что существуют poll и epoll)

Способ 2_2: select (и современная версия pselect())

Чтобы исправить ситуацию (ситуация про способ 2_1 polling: процессорное время тратится впустую на многократные и безрезультатные опросы), используют функцию select. Эта функция позволяет отслеживать состояние нескольких файловых дескрипторов (а в Unix к ним относятся и сокеты) одновременно.

Все, что касается функции select() объявляется в заголовочном файле <sys/select.h>.
что за fd_set

```
typedef struct {
      unsigned long fds_bits[__FD_SETSIZE / (8 * sizeof(long))];
```

```
} __kernel_fd_set;
typedef __kernel_fd_set       fd_set;
```

```
int select( int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout );
int pselect( int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, const struct timespec *timeout, sigset_t *sigmask );
```

Функция select() способна проверять состояние нескольких дескрипторов сокетов (или файлов) сразу.

Параметры:

- 1 - количество проверяемых дескрипторов (на единицу больше самого большого номера описателей из всех наборов)
- 2, 3, 4 - наборы дескрипторов, которые следует проверять, соответственно, на готовность к чтению, записи и на наличие исключительных ситуаций. В множество для чтения (2) записываются также слушающие сокеты *(видимо поэтому в листинге в самом начале туда закидывается один дескриптор)*
- 5 - можно указать интервал времени, по прошествии которого она вернет управление в любом случае.

Возвращаемое значение:

- значение больше нуля, которое соответствует числу готовых к операции дескрипторов. А все дескрипторы, которые привели к "срабатыванию" функции, записываются в соответствующие множества. Это позволяет нам проанализировать содержащиеся в множествах дескрипторы и выполнить над ними необходимые действия.

- ноль — в случае истечения тайм-аута,
- отрицательное значение при ошибке.

Сама функция select() – блокирующая, она возвращает управление, если хотя бы один из проверяемых сокетов готов к выполнению соответствующей операции.

Отличия:

- Вызов select() использует тайм-аут в виде struct timeval (с секундами и микросекундами), а в pselect() для этой же цели используется struct timespec (с секундами и наносекундами).
- Вызов select() может изменить значение параметра timeout, чтобы сообщить, сколько времени осталось, а вызов pselect() не поддерживает подобного поведения.
- Вызов select() не содержит параметра sigmask, и ведет себя как pselect() с параметром sigmask равным NULL. Если этот параметр для pselect() не равен NULL, то pselect() сначала замещает текущую маску сигналов на переданную в sigmask, а затем выполняет select() и восстанавливает исходную маску сигналов. Параметр тайм-аута может задаваться несколькими способами:
    - NULL, то есть время ожидания не ограничено;
    - ожидать конкретный указанный период времени;
    - не ожидать вообще (программный опрос, polling), когда структура тайм-аута инициализируется значением {0, 0}.

Кроме того, вводится понятие набора дескрипторов и макросы для работы с ними:

```
FD_CLR( int fd, fd_set *set ); // удаляет файловый дескриптор из набора
FD_ISSET( int fd, fd_set *set );//возвращает ненулевое значение, если указанный файловый дескриптор присутствует в наборе и ноль, если отсутствует
FD_SET( int fd, fd_set *set );//добавить файловый дескриптор в набор
FD_ZERO( fd_set *set );//очистить весь набор
```

Недостаток подхода: Программы, использующие неблокирующие сокеты вместе с select, получаются весьма запутанными. Если в случае с fork мы строим логику программы, как будто клиент всего один, здесь программа вынуждена отслеживать дескрипторы всех клиентов и работать с ними параллельно.
Достоинство: не создаются ни доп процессы, ни доп потоки. А создается множество соединений, которые обрабатываются в порядке поступления.

Пример
Вызов select() для проверки наличия входящих данных на сокете sock может выглядеть так:

```
fd_set set;
struct timeval interval;
FD_SET(sock, &set);
tv.tv_sec = 1;
tv.tv_usec = 500000;
...
select(1, &set, NULL, NULL, &tv);
if (FD_ISSET(sock, &set) {
// Есть данные для чтения
}
```

Способ 2_3: poll (и современная версия epoll()) (насколько я понимаю, техника polling вообще не связано с poll())

Как представляются дескрипторы

Отслеживаемый набор файловых дескрипторов представляет собой массив структур:

```
struct pollfd {
    int   fd;         /* файловый дескриптор */
    short events;     /* запрашиваемые события */
    short revents;    /* возвращённые события */
};
```

- В поле fd содержится файловый дескриптор открытого файла. Если значение поля отрицательно, то соответствующее поле events игнорируется, а полю revents возвращает ноль
- Поле events представляет собой входной параметр — битовую маску, указывающую на события, происходящие с файловым дескриптором fd, которые важны для приложения. Если это поле равно нулю, то возвращаемыми событиями в revents могут быть POLLHUP, POLLERR и POLLNVAL (смотрите ниже).
- Поле revents представляет собой параметр-результат, в который ядро помещает информацию о произошедших событиях. В revents могут содержаться любые битовые флаги из задаваемых в events, или там может быть одно из значений: POLLERR, POLLHUP или POLLNVAL. Эти три битовых флага не имеют смысла в поле events, но будут установлены в поле revents, если соответствующее условие истинно.

Сами функции

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec
*tmo_p, const sigset_t *sigmask);
```

- fds - отслеживаемый набор файловых дескрипторов (массив структур struct pollfd)
- nfds –количество элементов в массиве fds
- timeout указывается количество миллисекунд, на которые будет блокироваться poll() в ожидании готовности файлового дескриптора. Значение timeout, равное нулю, приводит к немедленному завершению poll(), даже если ни один файловый дескриптор не готов.

Вызов poll() выполняет сходную с select() задачу: он ждёт пока один дескриптор из набора файловых дескрипторов не станет готов выполнить операцию ввода-вывода.

Если ни одно из запрошенных событий с файловыми дескрипторами не произошло или не возникло ошибок, то poll() блокируется до их появления.

Вызов будет заблокирован пока:
- файловый дескриптор не станет готов;
- вызов не прервется обработчиком сигнала;
- не истечет время ожидания.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ:
- При успешном выполнении возвращается количество тех дескрипторов, для которых возникли события или ошибки
- Значение 0 означает, что время ожидания истекло, и нет готовых файловых дескрипторов.

- В случае ошибки возвращается -1, а errno устанавливается в соответствующее значение.

Вот возможные биты, которые могут быть установлены/получены в events и revents:

- POLLIN - Есть данные для чтения.
- POLLPRI - Есть срочные данные для чтения
- POLLOUT – Теперь запись возможна, но запись данных больше, чем доступно места в сокете или канале
- POLLRDHUP– Удалённая сторона потокового сокета закрыла соединение, или отключила запись в одну сторону.
- POLLERR – Состояние ошибки (возвращается только в revents; игнорируется в events).
- POLLHUP – Зависание (hang up, возвращается только в revents; игнорируется в events). Заметим, что при чтении из канала, такого как канал (pipe) или потоковый сокет, это событие всего-навсего показывает, что партнер закрыл канал со своего конца. Дальнейшее чтение из канала будет возвращать 0 (конец файла) только после потребления всех неполученных данных в канале.
- POLLNVAL – Неверный запрос: fd не открыт (возвращается только в revents; игнорируется в events).

Отношения между poll() и ppoll() аналогичны родству select(2) и pselect(2)

epoll

epoll - фабрика уведомлений о событиях ввода/вывода,описана в **<sys/epoll.h>**

epoll является вариантом poll(2), используемой либо как Edge, либо как Level Triggered интерфейс и хорошо масштабируемой для больших номеров наблюдаемых fds. Для настройки и управления epoll предлагаются три системных вызова:

- epoll подключается к описателю файла, созданного epoll_create(2).
- Интерес к соответствующим описателям файлов затем регистрируется через epoll_ctl(2).
- Наконец, фактическое ожидание запускается epoll_wait(2).

Примеры из лабораторной работы:

Вопросы с лабы:

- какой мультиплексор использовали?

select

- какие аргументы передаете мультиплексору?

в первом же коде сервера расписано

- почему мультиплексирование лучше, чем ожидание отдельного сокета?

Все соединения опрашиваются одновременно, сокращается время блокировки

Сравнение мультиплексирования и многопоточности с последних лаб (кидали фотку в группу): предпочесть мультиплексирование в потоке, когда количество потоков зависит от количества обслуживаемых вами клиентов.
Также было сказано: мультиплексированием это аналог многопоточности

- *как взаимодействуют сокеты AF_UNIX*

Через файловое пространство имен (через специальные файлы в файловой системе)

- сделайте чтобы файл сокета сервера и сокета клиента назывался одинаково(в коде)

не пересобираем иначе не сработает))

- что позволяет айпи INADR_ANY(в сетевых)?

Сервер зарегистрируется на всех адресах той машины, на которой он запускается. Реализовать на отдельной машине работу распределенной системы. Подключение к любому свободному айпи адресу

Привожу 5 вариантов на всякий. Но думаю так: Когда пишем про мультиплексирование – достаточно первого, Когда про сокеты – первого и третьего.

select (мультиплексирование); AF_INET, SOCK_STREAM

сервер

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

#define BUF_SIZE 256
#define PORT 3425
#define MAX_CLIENTS_NUMB 5


int sock_fd;


void del_socket(void)
{
    close(sock_fd);
}
```

```c
void sigtstp_handler(int signum)
{
    del_socket();
    exit(0);
}



void receive_from_clients(int *clients, fd_set *fd_readset)
{
    char buf[BUF_SIZE];
    int bytes;

    for (int i = 0; i < MAX_CLIENTS_NUMB; i++)
    {
        // FD_ISSET проверяет, является ли описатель частью набора

        if (FD_ISSET(clients[i], fd_readset))
        {
           // Поступили данные от клиента, читаем их
            bytes = recv(clients[i], buf, BUF_SIZE, 0);
           if (bytes <= 0)
            {
                // Соединение разорвано, удаляем сокет из
множества
                close(clients[i]);
                clients[i] = 0;
            }
            else if (bytes > 0)
            {
                buf[bytes] = 0;
                printf("Server recieved from client %d: %s\n", i,
buf);

                memset(buf, 0, BUF_SIZE);
                sprintf(buf, "Message recieved\n");
                send(clients[i], buf, sizeof(buf), 0);
            }
        }
    }
}

int main(void)
{
    struct sockaddr_in serv_addr;
```

```
    fd_set fd_readset;
    int clients[MAX_CLIENTS_NUMB] = {0};

    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd == -1)
    {
        printf("socket() failed\n");
        return EXIT_FAILURE;
    }

    signal(SIGTSTP, sigtstp_handler); //изменение обработчика
сигнала

    fcntl(sock_fd, F_SETFL, O_NONBLOCK);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY; //сервер
зарегистрируется на всех адресах той машины, на которой она
выполняется.
    serv_addr.sin_port = htons(PORT);

     if (bind(sock_fd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) == -1)
    {
      printf("bind() failed\n");
     del_socket();
      return EXIT_FAILURE;
    }

    if (listen(sock_fd, MAX_CLIENTS_NUMB) == -1)
    {
        printf("listen() failed\n");
        del_socket();
        return EXIT_FAILURE;
    }


    printf("Listening.\nPress Ctrl + Z to stop...\n");

    while(1)
    {
        // initialize the descriptor set to the null set.
       FD_ZERO(&fd_readset);
       // FD_SET - добавление дескриптора в набор
       // слушающий сокет добавляется в набор дескрипторов,
которые  следует проверять на готовность к чтению
       FD_SET(sock_fd, &fd_readset);
```

```
        int max_fd = sock_fd;
        for (int i = 0; i < MAX_CLIENTS_NUMB; i++)
        {
            if (clients[i])
            {
                FD_SET(clients[i], &fd_readset);
            }

            if (max_fd < clients[i])
           {
               max_fd = clients[i];
           }
         }


       // Ждём события в одном из сокетов
        if (select(max_fd + 1, &fd_readset, NULL, NULL, NULL) <=
0)
        {
            printf("select() failed\n");
            del_socket();
            return EXIT_FAILURE;
        }

       // Определяем тип события и выполняем соответствующие
действия
       // FD_ISSET проверяет, является ли описатель частью набора
       if (FD_ISSET(sock_fd, &fd_readset))
       {
          // Поступил новый запрос на соединение (sock_fd -
слушающий сокет)
           // блокируется, пока не появится "свисток" connect
(NULL - адрес и длина адреса запросившего, неважны)
                int new_sock = accept(sock_fd, NULL, NULL);

                if (new_sock == -1)
                {
                     printf("accept() failed\n");
                     del_socket();
                     return EXIT_FAILURE;
                }

                fcntl(new_sock, F_SETFL, O_NONBLOCK);

                int place_not_found = 1;
                for (int i = 0; i < MAX_CLIENTS_NUMB &&
```

```
place_not_found; i++)
                {
                        if (!clients[i])
                        {
                                clients[i] = new_sock;
                                printf("New connection %d\n", i);
                                place_not_found = 0;
                        }
                }
        }

        receive_from_clients(clients, &fd_readset);
    }
    return EXIT_SUCCESS;
}
```

Клиент

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <signal.h>

#define BUF_SIZE 256
#define PORT 3425
#define SOCK_ADDR "localhost"


int sock_fd;


void sigtstp_handler(int signum)
{
     printf("\nCatch SIGTSTP\n");
    if (close(sock_fd) == -1) //закрытие сокета
    {
        printf("close() failed\n");
        return;
```

```
    }
    exit(0);
}


int main(void)
{
     srand(time(NULL));

    struct sockaddr_in serv_addr;
    struct hostent *host;
    char message[BUF_SIZE];


    sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (sock_fd < 0)
    {
      printf("socket() failed\n");
      return EXIT_FAILURE;
    }

     signal(SIGTSTP, sigtstp_handler); //изменение обработчика
сигнала

     //преобразование доменного имени сервера в его сетевой адрес.
    host = gethostbyname(SOCK_ADDR);
    if (!host)
    {
        printf("gethostbyname() failed\n");
        return EXIT_FAILURE;
    }

    serv_addr.sin_family = AF_INET; // Семейство адресов
    //главное отличие sockaddr_in от sockaddr_un - наличие
параметра sin_port, предназначенного для хранения значения порта.
     serv_addr.sin_port = htons(PORT); // Номер порта
    serv_addr.sin_addr = *((struct in_addr*) host-
>h_addr_list[0]); // IP-адрес

    // Для установления активного соединения по переданному
адресу.
     if (connect(sock_fd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) == -1)
    {
          printf("connect() failed\n");
        return EXIT_FAILURE;
    }
```

```
    while(1)
    {
        memset(message, 0, BUF_SIZE);
        sprintf(message, "pid = %d\n", getpid());

        if (send(sock_fd, message, sizeof(message), 0) == -1)
        {
            printf("send() failed\n");
            return EXIT_FAILURE;
        }


           memset(message, 0, BUF_SIZE);
        int bytes = recv(sock_fd, message, BUF_SIZE, 0);

        if (bytes > 0)
        {
            message[bytes] = 0;
            printf("Client recieved from server: %s\n", message);
        }

        sleep(1 + rand() % 3);
    }

    close(sock_fd);

    return EXIT_SUCCESS;
}
```

epoll (мультиплексирование): на всякий

сервер

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>
#include <signal.h>
#include <netinet/in.h>

#include <sys/epoll.h>
#include <fcntl.h>
```

```c
#include <errno.h>

#define BUF_SIZE 256
#define PORT 8080
#define MAX_EPOLL_EVENTS 32
static int sFd;

void cleanup() {
    close(sFd);
}

void sighandler(int sig) {
    cleanup();
    printf("\nЗавершение работы...\n");
    exit(0);
}

int main() {
    sFd = socket(AF_INET, SOCK_STREAM, 0);
    if (sFd == -1) {
        printf("Socket error\n");
        return -1;
    }
    struct epoll_event events[MAX_EPOLL_EVENTS];
    int epfd = epoll_create(1);
    struct epoll_event event;
    event.events = EPOLLIN | EPOLLOUT;
    event.data.fd = sFd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, sFd, &event);
    struct sockaddr_in servaddr;
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);
    if (bind(sFd, (struct sockaddr *) &servaddr, sizeof(servaddr))
< 0) {
        printf("Bind error\n");
        cleanup();
        return -1;
    }

    char msgto[BUF_SIZE];
    char msgfrom[BUF_SIZE];

    if (signal(SIGINT, sighandler) == SIG_ERR) {
        perror("Signal error\n");
        cleanup();
```

```
        return -1;
    }

    struct sockaddr_un client;
    int client_len = sizeof(client);

    if (listen(sFd, 5) == -1) {
        perror("Listen error\n");
        cleanup();
        return -1;
    }

    printf("Сервер ждет сообщения\n(Нажмите Ctrl+C для звершения
работы)\n");
    for (;;) {
        int num_ready = epoll_wait(epfd, events, MAX_EPOLL_EVENTS,
1000/*timeout*/);
        if (num_ready == -1) {
            perror("Epoll_wait error\n");
            cleanup();
            return -1;
        }
        for (int i = 0; i < num_ready; i++) {
            if ((events[i].events & EPOLLERR) ||
                (events[i].events & EPOLLHUP) ||
                (!(events[i].events & EPOLLIN))) {
                perror("epoll error\n");
                close(events[i].data.fd);
            }
            else if (events[i].data.fd == sFd) {
                client_len = sizeof(client);
                int conn_socket = accept(sFd, (struct sockaddr *)
&client, &client_len);
                if (conn_socket == -1) {
                    if ((errno == EAGAIN) ||
                        (errno == EWOULDBLOCK)) {
                        break;
                    }
                    else {
                        perror ("accept error");
                        cleanup();
                        break;
                    }
                }
                event.data.fd = conn_socket;
                event.events = EPOLLIN;
```

```
                printf("Установлено соединение с сокетом %d\n",
conn_socket);
                if (epoll_ctl(epfd, EPOLL_CTL_ADD, conn_socket,
&event) == -1) {
                    perror("epoll_ctl error\n");
                    cleanup();
                    return -1;
                }
            }
            else {
                int bytes = recv(events[i].data.fd, msgfrom,
BUF_SIZE, 0);
                if (bytes == -1) {
                    perror("recvfrom error\n");
                    cleanup();
                    return -1;
                } else if (bytes == 0) {
                    if (epoll_ctl(epfd, EPOLL_CTL_DEL,
events[i].data.fd, &event) == -1) {
                        perror("epoll_ctl error\n");
                        cleanup();
                        return -1;
                    }
                    printf("Закрытие соединения с сокетом %d\n",
events[i].data.fd);
                    close(events[i].data.fd);
                } else {
                    msgfrom[bytes] = '\0';
                    printf("\nПолучено сообщение: %s\n", msgfrom);
                    snprintf(msgto, BUF_SIZE, "Привет, %s, будь
здоров!", msgfrom);
                    if (send(events[i].data.fd, msgto, BUF_SIZE,
0) == -1) {
                        perror("sendto error\n");
                        cleanup();
                        return -1;
                    } else {
                        printf("Отправлено сообщение: %s\n",
msgto);
                    }
                }
            }
        }
    }
}
```

клиент

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/epoll.h>
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <arpa/inet.h>

#define BUF_SIZE 256
#define PORT 8080
#define SERVER "127.0.0.1"
#define MAX_EPOLL_EVENTS 32
static int sFd;
void cleanup() {
    close(sFd);
}

void sighandler(int sig) {
    cleanup();
    printf("\nЗавершение работы...\n");
    exit(0);
}

int main() {
    sFd = socket(AF_INET, SOCK_STREAM, 0);
    if (sFd == -1) {
        printf("Socket error\n");
        return -1;
    }
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, SERVER, &serv_addr.sin_addr.s_addr) ==
0) {
        perror("inet_pton error\n");
        cleanup();
        return -1;
    }

    if (connect(sFd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) == -1 &&
            errno != EINPROGRESS) {
```

```
        perror("connect error\n");
        cleanup();
        return -1;
    }

    struct epoll_event events[MAX_EPOLL_EVENTS];
    int epfd = epoll_create(1);
    struct epoll_event event;
    event.events = EPOLLIN | EPOLLOUT | EPOLLRDHUP;
    event.data.fd = sFd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, sFd, &event);

    if (signal(SIGINT, sighandler) == SIG_ERR) {
        perror("Signal error\n");
        cleanup();
        return -1;
    }

    char msgto[BUF_SIZE];
    char msgfrom[BUF_SIZE];

    snprintf(msgto, BUF_SIZE, "процесс %d", getpid());
    int num_ready;
    for (;;) {
        num_ready = epoll_wait(epfd, events, MAX_EPOLL_EVENTS,
1000);
        if (num_ready == -1) {
            perror("Epoll_wait error\n");
            cleanup();
            return -1;
        }
        for (int i = 0; i < num_ready; i++) {
            if (events[i].events & EPOLLOUT) {
                if (send(events[i].data.fd, msgto, sizeof(msgto),
0) == -1) {
                    perror("Send error\n");
                    cleanup();
                    return -1;
                } else {
                    printf("Отправлено сообщение: %s\n", msgto);
                }
                int bytes;

                if ((bytes = recv(events[i].data.fd, msgfrom,
BUF_SIZE, 0)) == -1) {
                    perror("recv error");
```

```
                    cleanup();
                    return -1;
                } else {
                    msgfrom[bytes] = '\0';
                    printf("Получено сообщение: %s\n", msgfrom);
                }
            }
        }
        sleep(5);
    }
}
```

AF_UNIX, SOCK_DGRAM; с отправкой ответа от сервера клиенту, 2 сокета

сервер
(цикл while заблокирован по просьбе X)

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>

#define SOCKET_SERVER_NAME "server_socket"
#define BUF_SIZE 100


static int sfd;

void cleanup_socket(void)
{
    close(sfd);
    unlink(SOCKET_SERVER_NAME);
}

void sighandler(int signum)
{
     printf("\nCaught signal\n");
    cleanup_socket();
    exit(0);
}
```

```c
int main(void)
{
    if ((sfd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
     {
        perror("socket failed\n");
        return errno;
    }

    struct sockaddr_un sock_addr;
    sock_addr.sun_family = AF_UNIX;
    strcpy(sock_addr.sun_path, SOCKET_SERVER_NAME);

    if (bind(sfd, (struct sockaddr*)&sock_addr, sizeof(sock_addr))
== -1)
     {
        perror("bind failed\n");
        cleanup_socket();
        return errno;
    }

    signal(SIGINT, sighandler);


    char msg_rec[BUF_SIZE];
    char msg_send[BUF_SIZE];

    //while (1)
     //{
        printf("Waiting for client..Press Ctrl + C to stop\n");

        struct sockaddr_un client_addr;
        int addr_size;

        size_t bytes = recvfrom(sfd, msg_rec, BUF_SIZE, 0, (struct
sockaddr*) &client_addr, (socklen_t*)&addr_size);
        if (bytes == -1)
           {
            perror("recvfrom failed");
            cleanup_socket();
            return errno;
        }

           msg_rec[bytes] = '\0';
           printf("Server received message from %s: %s\n",
client_addr.sun_path, msg_rec);
```

```
        strcpy(msg_send, "Message was recieved: ");
        strcat(msg_send, msg_rec);
        bytes = sendto(sfd, msg_send, strlen(msg_send), 0, (struct
sockaddr*) &client_addr, addr_size);
        if (bytes <= 0)
           {
            printf("sendto failed");
            cleanup_socket();
            return errno;
        }
    //}
}
```

клиент
тут есть 2 варианта:

1. Все, что не закомментировано
2. Если использовать (раскомментировать) connect, то можно вместо sendto (закомментировать) использовать просто send (раскомментировать). Так как с помощью connect можно указать так сказать адрес всех отправок и приемов по умолчанию

При этом в обоих случаях достаточно использовать recv (а не recvfrom), так как адрес сервера после отправки и приема сообщения нам уже не важен

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>


#define SOCKET_SERVER_NAME "server_socket"
#define SOCKET_CLIENT_NAME "client_socket_"
#define BUF_SIZE 100


static int sfd;

void cleanup_socket(void)
{
```

```
      close(sfd);
    unlink(SOCKET_CLIENT_NAME);
}

void sighandler(int signum)
{
    cleanup_socket();
    exit(0);
}

int main(int argc, char* argv[])
{
    sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sfd == -1)
      {
         perror("socket failed");
         return errno;
    }


      signal(SIGINT, sighandler);

      char msg[BUF_SIZE];
    snprintf(msg, BUF_SIZE, "pid_%d", getpid());


    /* client socket */
    struct sockaddr_un addr_client;
    addr_client.sun_family = AF_UNIX;

    strcpy(addr_client.sun_path, SOCKET_CLIENT_NAME);
    strcat(addr_client.sun_path, msg);

    if (bind(sfd, (struct sockaddr*)&addr_client,
sizeof(addr_client)) == -1)
      {
         perror("bind failed");
         cleanup_socket();
         return errno;
      }

    /* server socket */
    struct sockaddr_un server_addr;
    server_addr.sun_family = AF_UNIX;
    strcpy(server_addr.sun_path, SOCKET_SERVER_NAME);
```

```
     /*
    if (connect(sfd, (struct sockaddr *) &server_addr,
sizeof(server_addr)) == -1)
     {
        perror("connect failed");
           cleanup_socket();
        return errno;
    }
     */


     size_t bytes;


     while (1)
     {
          bytes = sendto(sfd, msg, strlen(msg), 0, (struct
sockaddr*) &server_addr, sizeof(server_addr));
          if (bytes <= 0)
          {
               printf("sendto failed");
               cleanup_socket();
               return errno;
          }

          /*
          if (send(sfd, msg, strlen(msg), 0) == -1)
          {
               perror("send failed");
               return errno;
          }
          */


          char servermsg[BUF_SIZE] = {0};
          bytes = recv(sfd, servermsg, sizeof(servermsg), 0);
          if (bytes < 0)
          {
               perror("recv failed");
               return errno;
          }

          printf("Client received message: %s\n", servermsg);
          sleep(3);
     }
```

```
    cleanup_socket();
    return 0;
}
```

AF_UNIX, SOCK_DGRAM; без отправки ответа от сервера клиенту

сервер

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <signal.h>

#define SOCK_NAME "mysocket.soc"
#define BUF_SIZE 256

int sock_fd;

void del_socket(void)
{
    if (close(sock_fd) == -1) //закрытие сокета
    {
        printf("close() failed");
        return;
    }
    if (unlink(SOCK_NAME) == -1) //удаление файла сокета
    {
        printf("unlink() returned -1");
    }
}

void sigint_handler(int signum)
{
     printf("\nCatch SIGTSTP\n");
    del_socket();
    exit(0);
}

int main(void)
```

```
{
    struct sockaddr srvr_name;
    char buf[BUF_SIZE];
    int  bytes;

    if ((sock_fd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
    {
        printf("socket() failed");
        return EXIT_FAILURE;
    }

    srvr_name.sa_family = AF_UNIX;
    strcpy(srvr_name.sa_data, SOCK_NAME);

     //привязывает сокет к локальному адресу
    if (bind(sock_fd, &srvr_name, strlen(srvr_name.sa_data) +
sizeof(srvr_name.sa_family)) == -1)
    {
        printf("bind() failed\n");
        del_socket();
        return EXIT_FAILURE;
    }

    signal(SIGTSTP, sigint_handler); //изменение обработчика
сигнала

     //listen будет иметь смысл только для протоколов, нацеленных
на соединение (то есть только TCP)
     //поэтому здесь не используется
    printf("Waiting for messages.\nPress Ctrl + Z to stop...\n");

    while (1)
    {
           //блокирует программу до тех пор, пока на входе не
появятся новые данные.
           /*
           передается  указатель на еще одну структуру типа
sockaddr, в которой функция возвращает данные об адресе клиента,
           запросившего соединение (в случае файловых сокетов этот
параметр не несет полезной информации).
           Если информация об адресе клиента не нужна, то можно
передать значения NULL в предпоследнем и последнем параметрах.
           */
        bytes = recvfrom(sock_fd, buf, sizeof(buf),  0, NULL,
NULL);
        if (bytes < 0)
        {
```

```
            del_socket();
            printf("recvfrom() failed");
            return EXIT_FAILURE;
        }
        buf[bytes] = 0;
        printf("Server recieved: %s\n", buf);
    }

    del_socket();
    return EXIT_SUCCESS;
}
```

клиент

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

#define SOCK_NAME "mysocket.soc"
#define BUF_SIZE 256

int main(void)
{
     //домен соединения (семейство адресов) AF_UNIX: Сокеты в
файловом пространстве имен
     //тип SOCK_DGRAM -  определяют ненадежную  службу datagram
без установления логического соединения, когда пакеты могут
передаваться без сохранения логического порядка
    // протокол - по умолчанию
     //создаёт конечную точку соединения и возвращает файловый
дескриптор
     int sock_fd = socket(AF_UNIX, SOCK_DGRAM, 0);

    if (sock_fd == -1)
    {
        printf("socket() failed");
        return EXIT_FAILURE;
    }

     //общая структура адреса (адрес в данном случае -- имя файла)
    struct sockaddr srvr_name;
    srvr_name.sa_family = AF_UNIX;
    strcpy(srvr_name.sa_data, SOCK_NAME);
```

```
     char buf[BUF_SIZE];
    sprintf(buf, "pid %d", getpid());

     while (1)
     {
          // передача данных серверу. 4 параметр 0 -
дополнительные флаги, далее адрес сервера и его длина
          if (sendto(sock_fd, buf, strlen(buf), 0, &srvr_name,
strlen(srvr_name.sa_data) + sizeof(srvr_name.sa_family)) == -1)
          {
               printf("sendto() failed");
               close(sock_fd); //закрытие сокета
               return EXIT_FAILURE;
          }

          printf("Client sent: %s\n", buf);

          sleep(3);
     }

    close(sock_fd);
    return EXIT_SUCCESS;
}

//адрес клиента не играет никакой роли, поэтому можно без bind
(привязка сокета к локальному адресу)
```

AF_UNIX, SOCK_STREAM. Криво, плохо, не повторять!: своего рода мультиплексирование

(привожу на случай, если спросят – а вот как при AF_UNIX и SOCK_STREAM делать connect и accept????)

сервер

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <signal.h>
```

```c
#define SOCK_PATH "mysocket"
#define BUF_SIZE 100
#define N_CLIENTS 5


int sock_fd;

void del_socket(void)
{
    if (close(sock_fd) == -1) //закрытие сокета
    {
        return;
    }
    if (unlink(SOCK_PATH) == -1) //удаление файла сокета
    {
        return;
    }
}

void sigtstp_handler(int signum)
{
     printf("Catch SIGTSTP\n");
    del_socket();
    exit(0);
}

int main()
{
     int client_sock_fd;
     struct sockaddr_un sock_addr;

     signal(SIGTSTP, sigtstp_handler);

     sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
     if(sock_fd == -1)
     {
          printf("socket() failed\n");
          return EXIT_FAILURE;
     }

     sock_addr.sun_family = AF_UNIX;
     strcpy(sock_addr.sun_path, SOCK_PATH);

     if(bind(sock_fd, (struct sockaddr*)&sock_addr,
strlen(sock_addr.sun_path) + sizeof(sock_addr.sun_family)) != 0)
     {
```

```
        printf("bind() failed\n");
        del_socket();
        return EXIT_FAILURE;
    }

    if(listen(sock_fd, N_CLIENTS) != 0 )
    {
        printf("listen() failed\n");
        del_socket();
        return EXIT_FAILURE;
    }

    printf("Listening.\nPress Ctrl + Z to stop...\n");

    int bytes;
    char recv_buf[BUF_SIZE];
    char send_buf[BUF_SIZE];

    while (1)
    {
        unsigned int sock_len;
        struct sockaddr_un remote_sock_addr;

        if((client_sock_fd = accept(sock_fd, (struct
sockaddr*)&remote_sock_addr, &sock_len)) == -1 )
        {
            printf("Error on accept() call \n");
            return 1;
        }


        memset(recv_buf, 0, BUF_SIZE);
        memset(send_buf, 0, BUF_SIZE);
        bytes = recv(client_sock_fd, recv_buf, BUF_SIZE, 0);

        if(bytes > 0)
        {
            printf("Server received: %s\n", recv_buf);
            strcpy(send_buf, "Message recieved: ");
            strcat(send_buf, recv_buf);

            if(send(client_sock_fd, send_buf,
strlen(send_buf), 0) == -1 )
            {
                printf("send() failed\n");
            }
        }
```

```
            else
            {
                  printf("recv() failed. Waiting for new connection\
n");
                  break;
            }
      }

      del_socket();
      return 0;
}
```

клиент

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>

#define SOCK_PATH "mysocket"
#define BUF_SIZE 100


int sock_fd;
void sigtstp_handler(int signum)
{
      printf("Catch SIGTSTP\n");
    close(sock_fd);
    exit(0);
}


int main()
{

      int bytes;
      struct sockaddr_un server_sock_addr;
      char recv_msg[BUF_SIZE];
      char send_msg[BUF_SIZE];

      memset(recv_msg, 0, BUF_SIZE);
      memset(send_msg, 0, BUF_SIZE);
```

```
      signal(SIGTSTP, sigtstp_handler);

      server_sock_addr.sun_family = AF_UNIX;
      strcpy(server_sock_addr.sun_path, SOCK_PATH);
      memset(send_msg, 0, BUF_SIZE);
      sprintf(send_msg, "pid %d", getpid());

      while (1)
      {
            if((sock_fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
            {
                  printf("socket() failed\n");
                  return EXIT_FAILURE;
            }


            if(connect(sock_fd, (struct
sockaddr*)&server_sock_addr, strlen(server_sock_addr.sun_path) +
sizeof(server_sock_addr.sun_family)) == -1)
            {
                  printf("connect() failed %d\n", errno);
                  close(sock_fd);
                  return EXIT_FAILURE;
            }


            if(send(sock_fd, send_msg, strlen(send_msg), 0 ) == -1)
            {
                  printf("send() failed\n");
            }


            memset(recv_msg, 0, BUF_SIZE);

            if((bytes = recv(sock_fd, recv_msg, BUF_SIZE, 0)) > 0 )
            {
                  printf("Client received: %s\n", recv_msg);
            }
            else
            {
                  if(bytes < 0)
                  {
                        printf("recv() failed\n");
                  }
                  else
                  {
                        printf("Server socket closed \n");
```

```
                    close(sock_fd);
                    break;
                }

            }
            sleep(3);
            close(sock_fd);
        }

        return 0;
}
```

AF_UNIX, SOCK_STREAM | SOCK_NONBLOCK: То же, что и в предыдущем, но еще и с флагом SOCK_NONBLOCK

Сервер

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <signal.h>

#define SOCK_PATH "mysocket"
#define BUF_SIZE 100
#define N_CLIENTS 5


int sock_fd;

void del_socket(void)
{
    if (close(sock_fd) == -1) //закрытие сокета
    {
        return;
    }
    if (unlink(SOCK_PATH) == -1) //удаление файла сокета
    {
        return;
    }
}

void sigtstp_handler(int signum)
{
     printf("Catch SIGTSTP\n");
```

```
    del_socket();
    exit(0);
}

int main()
{
     int client_sock_fd;
     struct sockaddr_un sock_addr;

     signal(SIGTSTP, sigtstp_handler);

     sock_fd = socket(AF_UNIX, SOCK_STREAM | SOCK_NONBLOCK , 0);
     if(sock_fd == -1)
     {
          printf("socket() failed\n");
          return EXIT_FAILURE;
     }

     sock_addr.sun_family = AF_UNIX;
     strcpy(sock_addr.sun_path, SOCK_PATH);

     if(bind(sock_fd, (struct sockaddr*)&sock_addr,
strlen(sock_addr.sun_path) + sizeof(sock_addr.sun_family)) != 0)
     {
        printf("bind() failed\n");
        del_socket();
        return EXIT_FAILURE;
     }

     if(listen(sock_fd, N_CLIENTS) != 0 )
     {
        printf("listen() failed\n");
        del_socket();
        return EXIT_FAILURE;
     }

     printf("Listening.\nPress Ctrl + Z to stop...\n");

     int bytes;
     char recv_buf[BUF_SIZE];
     char send_buf[BUF_SIZE];

     while (1)
     {
          unsigned int sock_len;
          struct sockaddr_un remote_sock_addr;
```

```
          if((client_sock_fd = accept(sock_fd, (struct
sockaddr*)&remote_sock_addr, &sock_len)) == -1 )
          {
               continue;
               //printf("Error on accept() call \n");
               //return 1;
          }


          memset(recv_buf, 0, BUF_SIZE);
          memset(send_buf, 0, BUF_SIZE);
          bytes = recv(client_sock_fd, recv_buf, BUF_SIZE, 0);

          if(bytes > 0)
          {
               printf("Server received: %s\n", recv_buf);
               strcpy(send_buf, "Message recieved: ");
               strcat(send_buf, recv_buf);

               if(send(client_sock_fd, send_buf,
strlen(send_buf), 0) == -1 )
               {
                    printf("send() failed\n");
               }
          }
          else
          {
               continue;
               //printf("recv() failed. Waiting for new
connection\n");
               //break;
          }
     }

     del_socket();
     return 0;
}
```

Клиент

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
```

```c
#include <signal.h>
#include <errno.h>

#define SOCK_PATH "mysocket"
#define BUF_SIZE 100


int sock_fd;
void sigtstp_handler(int signum)
{
     printf("Catch SIGTSTP\n");
    close(sock_fd);
    exit(0);
}


int main()
{

     int bytes;
     struct sockaddr_un server_sock_addr;
     char recv_msg[BUF_SIZE];
     char send_msg[BUF_SIZE];

     memset(recv_msg, 0, BUF_SIZE);
     memset(send_msg, 0, BUF_SIZE);
     signal(SIGTSTP, sigtstp_handler);

     server_sock_addr.sun_family = AF_UNIX;
     strcpy(server_sock_addr.sun_path, SOCK_PATH);
     memset(send_msg, 0, BUF_SIZE);
     sprintf(send_msg, "pid %d", getpid());

     while (1)
     {
          if((sock_fd = socket(AF_UNIX, SOCK_STREAM |
SOCK_NONBLOCK , 0)) == -1 )
          {
               printf("socket() failed\n");
               return EXIT_FAILURE;
          }


          if(connect(sock_fd, (struct
sockaddr*)&server_sock_addr, strlen(server_sock_addr.sun_path) +
sizeof(server_sock_addr.sun_family)) == -1)
          {
```

```
                printf("connect() failed %d\n", errno);
                close(sock_fd);
                return EXIT_FAILURE;
        }


        if(send(sock_fd, send_msg, strlen(send_msg), 0 ) == -1)
        {
                printf("send() failed\n");
        }


        memset(recv_msg, 0, BUF_SIZE);


    while (1)
    {
        if((bytes = recv(sock_fd, recv_msg, BUF_SIZE, 0)) > 0 )
        {
                printf("Client received: %s\n", recv_msg);
                break;
        }
        else
        {
                if(bytes < 0)
                {
                        continue;
                        //printf("recv() failed\n");
                }
                else
                {
                        printf("Server socket closed \n");
                        close(sock_fd);
                        break;
                }

        }
    }
        sleep(3);
        close(sock_fd);
    }

    return 0;
}
```

OFFTOP

Сравнение семейств сокетов

Общее - взаимодействие по модели клиент-сервер. Есть сервер, его функция - предоставлять сервис, клиенты обращаются к серверу, чтобы его получить. Обычно клиентов много, сервер один.

Но среды совсем разные

Разница. AF_UNIX – сокеты в файловом пространстве имен. Здесь нет реальных адресов (типа номер порта и сетевой адрес), взд – через спец файл. Отсюда функции, которые необходимо использовать на стороне клиента и сервера.

В целом: функции, определенные на сокетах

Объявления (обратить внимание: используется общая структура sockaddr, а не sockaddr_in):

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen)
int listen(int sockfd, int backlog); //адрес не важен. Имеет смысл только для протокола TCP, ориентированного на соединение.
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
//serv_addr - подчеркивается, что выполняется клиентом, и клиент должен указать адрес сервера
int accept(int sockfd, struct sockaddr *cli_addr, int *addrlen);
// 1) последний аргумент с типом socklen_t=int, 2)подчеркивается, что адрес клиента
```

1. socket(2) создаёт сокет,

После создания с помощью socket(2), сокет появляется в адресном пространстве (семействе адресов), но без назначенного адреса. bind() назначает адрес, заданный в addr, сокету, указываемому дескриптором файла sockfd. В силу традиции, эта операция называется «присваивание сокету имени».

2. bind(2) привязывает сокет к локальному адресу,

Сокет нужно связать с некоторым адресом (очень сильно зависит от семейства). Bind используется для назначения сокету локального адреса.

Для семейства AF_UNIX – это файл. То есть сокеты этого семейства взаимодействуют через файловую подсистему. Для сокетов интернета адрес состоит из IP адреса сетевого интерфейса локальной системы и номера порта.

Клиенты даже в случае сетевых сокетов могут действовать и без bind, так как их адрес не играет никакой роли. Если этот вызов не осуществляется, то адрес назначается автоматически.

А что если сервер заинтересован в отправке клиенту ответного сообщения? тоже пофиг на адрес?

3. listen(2) сообщает сокету, что должны приниматься новые соединения,

listen (только на стороне сервера) - указывает операционной системе, что сокет переходит в режим так называемого пассивного прослушивания **СОЕДИНЕНИЙ.**

listen будет иметь смысл только для протоколов, нацеленных на соединение (то есть только TCP)

4. connect(2) соединяет сокет с удалённым сокетным адресом,

connect (только на стороне клиента). Для установления активного соединения по переданному адресу.

Для протоколов без установленного соединения (UDP например) connect может использоваться, чтобы указать адрес назначения для всех передаваемых пакетов. (и тогда вместо recvfrom/sendto с указанием адреса можно использовать просто recv/send)

5. accept(2) используется для получения нового сокета для нового входящего соединения.

Accept – вызывается сервером, чтобы принять соединение при условии, что ранее сервер получил запрос соединения. До получения запроса соединения accept заблокирован.

Как функция может быть заблокирована? нужен свисток (то есть connect). Когда соединение принимается (свисток пришел), accept действует так: создает копию исходного сокета, чтобы этот исходный сокет оставался в состоянии listen для получения запросов, а новый сокет – в состоянии CONNECTED.

То есть вызов accept возвращает новый дескриптор файла для второго и тд сокетов. (поэтому в примере создается массив дескрипторов).

Такое дублирование сокетов при соединениях позволяет серверу принимать новые соединения без необходимости закрывать предыдущие.

6. send(2), sendto(2) и sendmsg(2) отправляют данные в сокет,
7. recv(2),recvfrom(2) и recvmsg(2) принимают данные из сокета.
8. для чтения и записи данных могут использоваться стандартные операции ввода-вывода: write(2),writev(2), sendfile(2), read(2) и readv(2). (но рекомендуется использовать sendto/recvfrom, потому что там указывается адрес)
9. poll(2) и select(2) ожидают поступления данных или готовятся к передаче данных. Пользователь может подождать наступления различных событий через poll() или select().

Вызовы ниже описаны в методе, на семинаре не разбирали вообще

Вызов socketpair(2) возвращает два соединённых анонимных сокета (реализовано только для некоторых локальных семейств, например AF_UNIX).

Вызов getsockname(2) возвращает адрес локального сокета, а getpeername(2) возвращает адрес удалённого сокета. Вызовы getsockopt(2) и setsockopt(2) используются для установки или считывания параметров протокола или уровня сокетов. Вызов ioctl(2) может быть использован для установки или чтения некоторых других параметров.

Вызов close(2) используется для закрытия сокета. Вызов shutdown(2) закрывает части полнодуплексного сокетного соединения.

Перемещение (seeking), или вызовы pread(2) и pwrite(2) с ненулевой позицией, для сокетов не поддерживается.

Для сокетов возможно создание неблокирующего ввода/вывода путем установки в файловый дескриптор сокета флага O_NONBLOCK с помощью вызова fcntl(2). При этом все блокировавшие раньше операции, будут возвращать EAGAIN (операция должна быть повторена позднее); connect(2) возвратит ошибку EINPROGRESS.

События ввода-вывода		
Событие	*Флаг poll*	*Когда происходит*
Чтение	*POLLIN*	*Поступили новые данные*
Чтение	*POLLIN*	*Установка соединения выполнена (для сокетов, ориентированных на соединение)*
Чтение	*POLLHUP*	*Другая сторона инициировала запрос на разъединение*
Чтение	*POLLHUP*	*Соединение разорвано (только для протоколов, ориентированных на соединение). Если производится запись в сокет, то также посылается сигнал SIGPIPE*
Запись	*POLLOUT*	*Сокет имеет достаточно места в буфере отправки для записи в него новых данных*
Чтение/ Запись	*POLLIN| POLLOUT*	*Исходящий вызов connect(2) завершён*
Чтение/ Запись	*POLLERR*	*Произошла асинхронная ошибка*
Чтение/ Запись	*POLLHUP*	*Другая сторона закрыла (shut down) одно направление*
Исключение	*POLLPRI*	*Пришли неотложные данные. При этом посылается сигнал SIGURG*

В ядре существует возможность информировать приложение о событиях с помощью сигнала SIGIO, что является альтернативой вызовам poll(2) и select(2). Для этого необходимо установить с помощью fcntl(2) в файловом дескрипторе сокета флаг O_ASYNC, а также назначить с помощью sigaction(2) корректный обработчик сигнала SIGIO.

Что еще

- еще раз: Порт – это адрес!
- 2 адресных пространства: ввода-вывода (64 кбайт - 1 сегмент 16 разрядной машины) и физической памяти. Причем они пересекаются. В результате по шине адреса может передаваться что угодно. Чтобы разделить эту адресацию, в архитектуре процессоров intel были включены южный и северный мосты, на которых реализована функция bus_mastering. В современных системах это осталось неизменным. Южный и северный мосты и bus_mastering заменены единой микросхемой.

Все про (аппаратные) прерывания:

Аппаратные прерывания (interrupts)

- это прерывания, поступающие от устройств (от таймера, от устройства ввода вывода и т.д.) (поступают от контроллера прерываний)
- это асинхронные события в системе: происходят вне зависимости от какой-либо работы, выполняемой процессором
- Примеры:
    i. Всегда отдельно рассматривается прерывание от системного таймера – особое и единственное периодическое прерывание с важнейшими системными функциями (а также единственное быстрое) *(В системах разделения времени – декремент кванта)*
    ii. Прерывание от внешнего устройства по завершении операции ввода/вывода – внешние устройства информируют процессор о том, что ввод/вывод завершен и процесс может перейти к обработке. При этом (даже вывод) происходит получение данных об успешности или неуспешности завершения операции.
    iii. Отдельно рассматривается – прерывание от действий оператора (win: ctrl+alt+delete, unix :ctrl+C)
- Бывают (У них разные входы: вход INTR (от INTerrupt Request — запрос на прерывание) и вход NMI (от Not Maskable Interrupt — немаскируемое прерывание):
    i. Маскируемые - прерывания, которые м.б. отложены или запрещены
    ii. Немаскируемые - невозможно запретить или отложить

Запрос прерывания и линии IRQ

IRQ(Interrupt Request) - запрос на обработку прерывания;

В конце каждой выполняемой команды процессор проверяет наличие прерывания на ножке процессора. Если сигнал был получен, то процесс переходит к исполнению обработчика прерывания - это процесс обработчика аппаратного прерывания.

В трёх-шинной архитектуре внешними устройствами управляют контроллеры или адаптеры.

Контроллером называется устройство которое как правило входит в состав внешнего устройства. Адаптером называется устройство, которое как правило находится на материнской плате.

По завершению операции процессор информируют специальные устройства - **контроллеры.**

Контроллер – программно-управляемое устройство, в нем имеется набор регистров и некоторая логика.

Контроллер получает от процессора команду, выполняя которую, контроллер берет на себя управление операцией ввода/вывода. По завершении операции ввода/вывода контроллер посылает на вход контроллера прерываний сигнал.

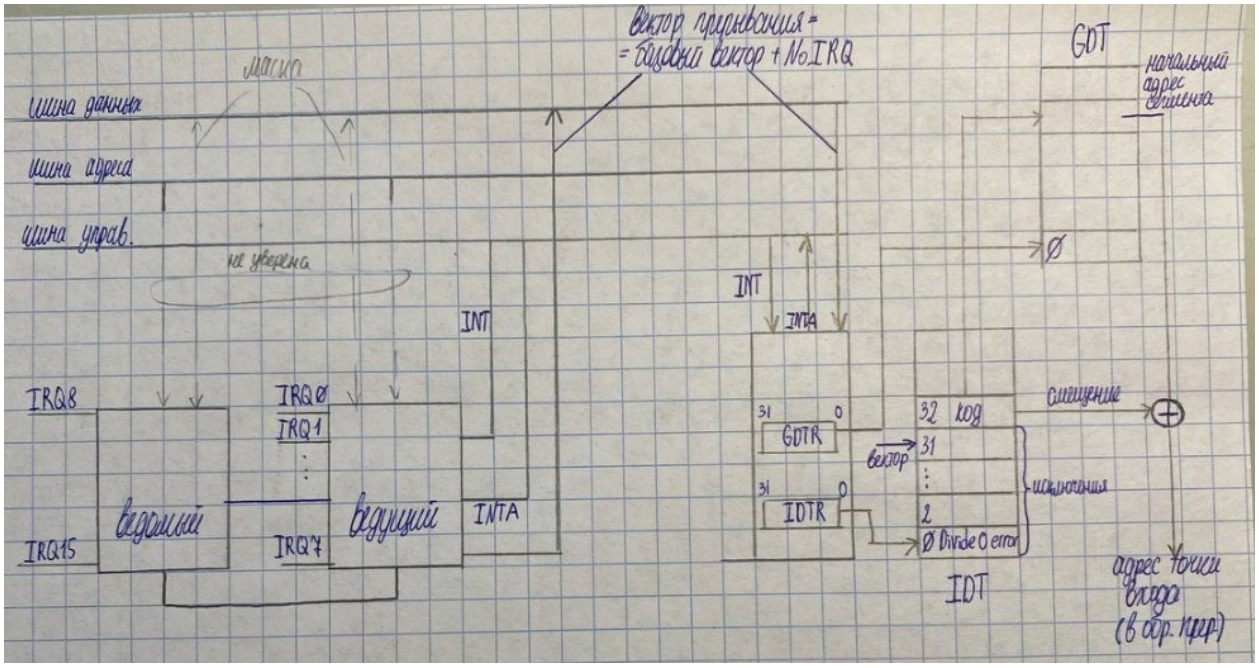
ЧТО ТАКОЕ ЛИНИЯ IRQ

Ответ в схеме!!!!!!
Если взять простейшую схему соединения внешних устройств с контроллером прерывания, **выделенная линия irq – та линия, на которую устройство посылает сигнал прерывания.** В старых системах каждому устройству выделялась конкретная линия – таймеру, клаве, принтеру. В досе это было видно. Эти понятия остались. **Это линия, которая передает сигнал прерывания**. И когда смотри proc/interrupts, 12 - мышь psпополам, x-таймер,y- клава, остальные – назначаются динамически, но суть та же.

Линия IRQ - контакт контроллера

Адресация аппаратных прерываний в з-р (трехшинная архитектура).

(если прерывание не замаскировано)



тут “32 код” = “32 сегмент кода”

(точка входа)

IDTR-32 разрядный регистр, который содержит начальный линейный адрес IDT (все эти регистры есть в каждом ядре).

На вход контроллера приходит сигнал, контроллер формирует сигнал int, который по шине управления переходит на ножку процессора. Процессор посылает intA по шине управления в контроллер. Контроллер выставляет на шину данных вектор прерывания, который содержит селектор к IDT.

Вектор поступает в процессор. Процессор берет значение базового адреса IDT из регистра IDTR и по селектору (из вектора прерывания) находит нужный дескриптор.

Этот дескриптор содержит селектор для сегмента кода, смещение к точке входа и атрибуты. По селектору выбираем дескриптор из GDT, берём оттуда базовый адрес сегмента и прибавляем его к смещению. Получаем линейный адрес точки входа

Отдельно адресуются ведущий и ведомый контроллеры.

Зубков

Существует два контроллера прерываний.

Первый контроллер, обслуживающий запросы на прерывания от IRQO до IRQ7, управляется через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты OAOh и OAlh. если несколько прерываний происходят одновременно, обслуживается в первую очередь то, у которого высший приоритет. При инициализации контроллера высший приоритет имеет IRQO (прерывание от системного таймера), а низший - IRQJ7. Все прерывания второго контроллера (IRQ8 - IRQ15) оказываются в этой последовательности между IRQ1 и IRQ3, так как именно IRQ2 используется для каскадирования этих двух контроллеров. В тот момент, когда выполняется обработчик аппаратного прерывания, других прерываний с низшими приоритетами нет, даже если обработчик выполнил команду sti. Чтобы разрешить другие прерывания, каждый обработчик обязательно должен послать команду EOI - конец прерывания - в соответствующий контроллер..

еще инфа

В ЗР для адресации прерывания имеется специальная таблица IDT. Если первые 32 исключения, и мы возьмем 8, то попадем на double fault. Чтобы адресовать прерывания, надо перепрограммировать контроллер на: ведущий-на базовый вектор 32, и этот номер использовать для обращения к таблице. Базового адреса нет, есть смещение и селектор. Отдельно адресуются ведущий и ведомый. От контроллера они могут получить маску??

В РР процессор использует вектор и таблицу векторов прерываний. У ведущего контроллера базовый вектор=8 (8+0=8h) и номер используется для получения смещения к адресу в таблице векторов прерываний. В DOS адрес наз. вектор (4 байт)

В з-р для каждого сегмента программы должен быть определен дескриптор - 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики. В р-р сегменты определяются их линейными адресами. (Р-Ф).

Таблица дескрипторов прерываний (IDT)

IDT (interrupt descriptor table) - системная таблица, предназначенная для хранения адресов обработчиков прерываний. (нужна для адресации обработчиков прерываний)

Первые 32 элемента таблицы - под исключения (синхронные события в процессе работы программы) (внутренние прерывания процессора) (в 386 всего 19 исключений (0-19), остальные (20-31) зарезервированы, а *в 486 – и того меньше (реально-18, остальные-зарезервированы)*

32-255 – определяются пользователем (user defined)

Смещение=номер исключения*8

Нам надо адресовать 2 обработчика – таймера и клавиатуры (аппаратные прерывания) через программирование контроллера прерываний. Сейчас прерывания как MSI (message signal interrupts)

- 0-divide error (ошибка деления на 0)
- 8-double fault (если выполнить исключение или маскируемое/немаскрируемое прерывание и возникла ошибка (паника…), завершается работа компьютера)
- 11-segment not present (сегмент отсутствует – надо выполнить определенные действия, чтобы сделать сегмент доступным. Касается управления памятью (нашей программе-не очень))
- 13-general protection (общая защита, должно быть обработано специальным образом. На все исключения-заглушки (double fault-не искл.), а на 13-специальная заглушка (у РФ отражено в структуре таблицы дескрипторов прерываний-без dup))
  (нарушение общей защиты (нарушение, код ошибки-та команда), происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. Номером)
- 14-page fault (fault переводится как исключение, но по-русски здесь прерывание) (страничное прерывание) – обращение к команде/данным, отсутствующим в программе – система должна загрузить нужную страницу. В CR2 адрес, на котором произошло прерывание)

Вектор	Название исключения	Класс исключения	Код ошибки	Команды, вызывающие исключение
0	Ошибка деления	Нарушение	Нет	div, idiv
1	Исключение отладки	Нарушение /ловушка	Нет	Любая команда
2	Немаскируемое прерывание			
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into
5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
7	Сопроцессор недоступен	Нарушение	Нет	esc, wait
8	Двойное нарушение	Авария	Да	Любая команда
9	Выход сопроцессора из сегмента(80386)	Авария	Нет	Команда сопроцессора с обращением к памяти
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страничное нарушение	Нарушение	Да	Команда обращения к памяти
15	Зарезервировано			
16	Ошибка сопроцессора	Нарушение	Нет	esc, wait
17	Ошибка выравнивания	Нарушение	Да	Команда обращения к памяти
18...31 Зарезервированы				
32...255 Предоставлены пользователю для аппаратных прерываний и команд int				

Типы шлюзов.

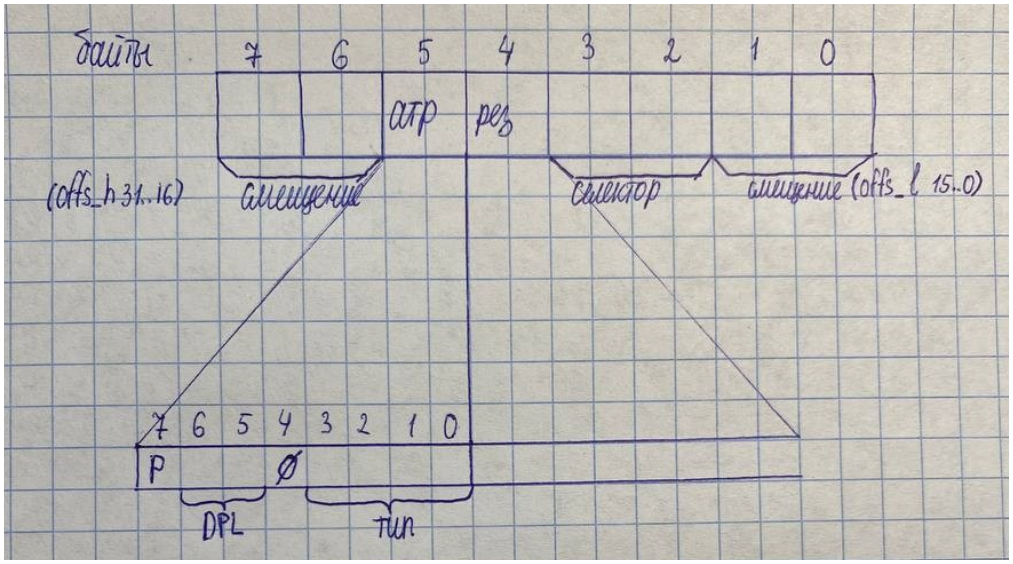
1. Ловушки (trap gate) - обработка исключений и программных прерываний (системных вызовов).

2. Прерывания (interrupt gate) - обработка аппаратных прерываний.

3. Задач (task gate)- *переключение задач в многозадачном режиме.*

Формат дескриптора прерывания

Формат дескриптора (шлюза) для IDT (в скобках – из учебника)



- Байты 0-1 (offs_1), 6-7 (offs_h): 32-битное смещение обработчика
- Байты 2-3 (sel): селектор (сегмента команд) (итого полный 3-хсловный адрес обработчика селектор: смещение)
- Байт 4 зарезервирован
- Байт 5: байт атрибутов - как в дескрипторах памяти за исключением типа:

  Типы: назначение:
  - 0-не определен
  - 1-свободный сегмент состояния задачи TSS 80286
  - 2-LDT
  - 3-занятый сегмент состояния задачи TSS 80286
  - 4-шлюз вызова Call Gate 80286
  - 5-шлюз задачи Task Gate
  - 6-шлюз прерываний Interrupt Gate 80286
  - 7-шлюз ловушки Trap Gate 80286
  - 8-не определен
  - 9- свободный сегмент состояния задачи TSS 80386+
  - Ah-не определен
  - Bh- занятый сегмент состояния задачи TSS 80386+
  - Ch-шлюз вызова Call Gate 80386+
  - Dh- не определен
  - Eh-шлюз прерываний Interrupt Gate 80386+
  - Fh- шлюз ловушки Trap Gate 80386+

Может принимать 16 значений, но в IDT допустимо 5: 5(задачи), 6(прерываний 286), 7(ловушки 286), Eh(прерываний 3/486), Fh(ловушки 3/486) (это по РФ)

  - 4-0 (а вообще это S - system (0 - системный объект, 1 - обычный)
  - 5-6-DPL - уровень привилегий (0 - уровень привилегий ядра, 3 - пользовательский/приложений, *1-2 - не используется в системах общего назначения*)

- 7-1 (а вообще это P - бит присутствия (1 - если сегмент в оперативной памяти, 0 - иначе)

Пример заполнения IDT из лабораторной работы.

```
;Структура idescr для описания дескрипторов (шлюзов) прерываний
idescr struc
    offs_l  dw 0
    sel     dw 0
    cntr    db 0
    attr    db 0
    offs_h  dw 0
idescr ends
```

```
;Таблица дескрипторов прерываний IDT
;Дескриптор: <offs_l, sel, rsrv, attr, offs_h>
;смещение позже?, селектор 32-разрядного сегмента кода
idt label byte

; Первые 32 элемента таблицы - под исключения-внутренние прерывния процессора
;(реально-18, остальные-зарезервированы)
;attr=8Fh: тип=ловушка 386/486(обр. программные пр. и искл., IF не меняется),
;системный объект, УП ядра, P=1
idescr_0_12 idescr 13 dup (<0,code32s,0,8Fh,0>)
; исключение 13 - нарушение общей защиты (нарушение, код ошибки-та команда)
; происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. номером
idescr_13 idescr <0,code32s,0,8Fh,0>
idescr_14_31 idescr 18 dup (<0,code32s,0,8Fh,0>)


; Затем 16 векторов аппаратных прерываний,
;attr=8Eh: тип=прерывание 386/486(обр. аппаратные пр., IF сбрасывается a iret восстанавливает),
;системный объект, УП ядра, P=1
;Дескриптор прерывания от таймера
int08 idescr <0,code32s,0,8Eh,0>
int09 idescr <0,code32s,0,8Eh,0>

idt_size = $-idt        ;размер
ipdescr df 0            ;псевдодескриптор
ipdescr16 dw 3FFh, 0, 0 ;содержимое регистра IDTR в РР: с адреса 0, 256*4=1кб=2^10
```

```
; Заносим в дескрипторы прерываний (шлюзы) смешение обработчиков прерываний.
lea eax, es:except_13
mov idescr_13.offs_l, ax
shr eax, 16
mov idescr 13.offs h, ax
```