1. 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. Мультиплексирование. Пример мультиплексирования для сокетов AF_INET, SOCK_STREAM. Сетевой стек. Пример (лаб. раб.)

1.1. Классификация типов ввода-вывода с точки зрения программиста

Идея распараллеливания функций

С оперативной памятью мы работаем при помощи команды mov, а с внешними устройствами - через порты ввода вывода командами in и out. (memory mapping, io mapping)

60-е - начало 70-х - появление интегральных микросхем, позволивших значительно увеличить мощности вычислительной системы, уменьшить её габариты.

В машинах ibm360,370 была реализована идея распараллеливания функцийй.

Так как процессор всегда был самым быстродействующим устройством системы и это стало их отличительной чертой. Внешние устройства - механика, медленные устройства.

Чтобы более эффективно использовать процессорное время, была реализована архитектура с распараллеливанием функций: в состав ibm360 были включены специальные процессоры-каналы, которые взяли на себя функцию усправления внешними устройствами.

Таким образом, процессор был освобождён от усправления внешними устройствами, но это требовало реализации в системе полноценной системы прерываний: процессор инициализирует ввода-выывода, а управляет ей канал. По завершении операции ввода-вывода прерывание информирует провессор о завершении данной операции - это основа функционирования любой системы.

В ПК на базе intel - шинная архитектура.

Архитектура мейнфреймов ibm называется канальной.

В шинной архитектуре внешними устройствами управляют специальные устройства - контроллеры.

Основное отличие асинхронного ввода-вывода от ввода-вывода, управляемого сигналом: при вводе-выводе, управляемым сигналом, сигнал информирует процесс и готовности данных, а при асинхронном вводе-выводе процесс вообще не блокируется и информируется о полном завершении операции ввода-вывода.

1.1.1. Блокирующий ввод-вывод (blocking)

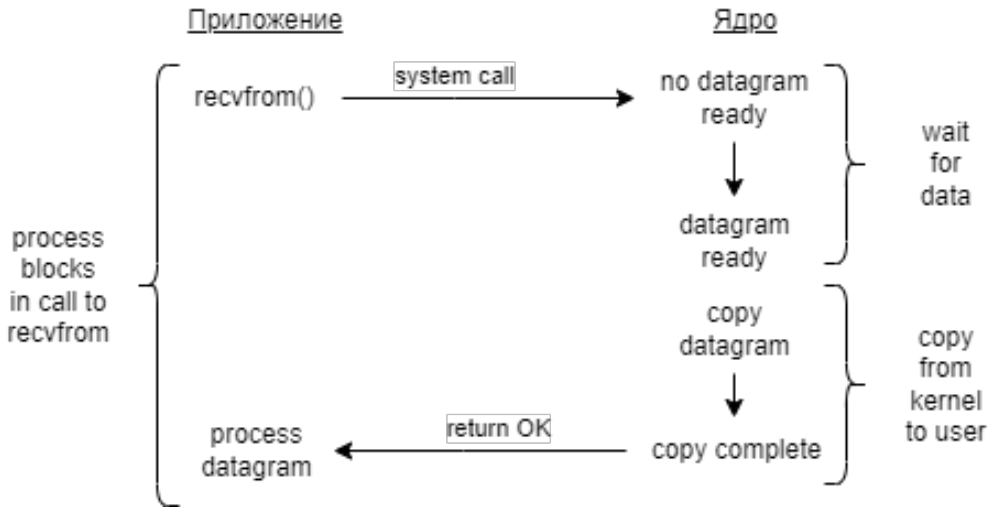
Блокирующий ввод-вывод (запрос на получение данных read). Используется ключевое слово receive from – общее средство передачи данных в системе. Передача данных с помощью сообщений – самый общий способ, который может рассматриваться как на отдельно стоящей машине, так и в распределенной системе.

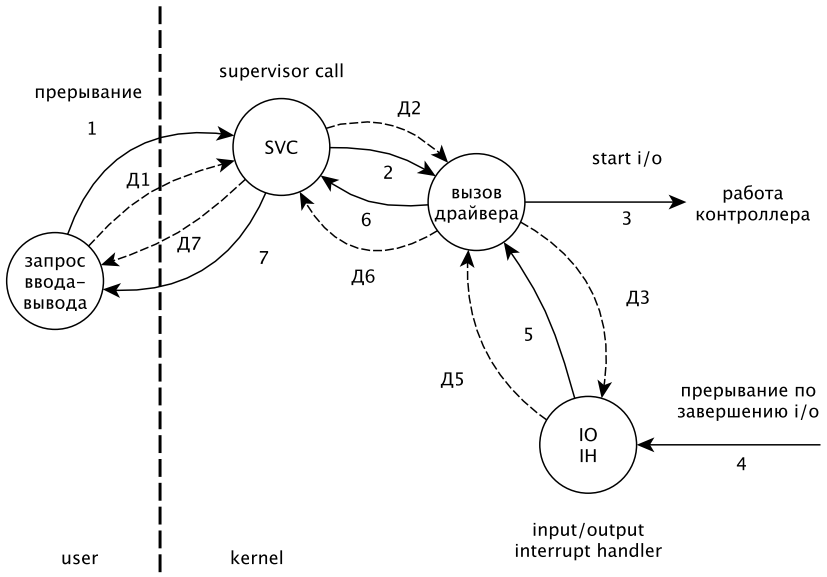
receive from = запрос на получение данных = read. Все сводится к системным вызовам read/write.

send to/receive from, как и read/write, переводят систему в режим ядра, так как это системные вызовы.

Процесс, выполнивший системный вызов ввода-вывода (не важно, read или write), будет ожидать ошибки, либо значение, соответствующее успешному выполнению системного вызова.

То есть процесс в любом случае получит данные (в случае read/receive from это особенно явно).





Когда приложение выполняет запрос ввода-вывода (в примере ввод – receive from), процесс блокируется до тех пор, пока не получит данные.

Что значит datagram ready (данные готовы)? Если мы вводим символ с клавиатуры, то он готов тогда, когда он введен в буфер клавиатуры.

Далее он копируется в буфер ядра, и уже оттуда в буфер приложения, после чего оно сможет продолжить свое выполнение.

Системы реального времени работают в готовыми данными (с датчиков из измерительных приборов, считавших показания). Данные готовы, их нужно только

получить. Если данные не готовы, будет возвращена ошибка, то есть мы не можем обработать соответствующие данные, и надо выполнить доп. действия. Надо понимать, когда это возможно, а когда нет.

Диаграмма из 1 вопроса на запрос ввода вывода + пояснения

Таким образом, весь ввод – вывод который мы реализовывали в своих программах, является блокирующим (пока процесс не получит данные, он будет заблокирован).

Такой ввод-вывод возможен, только если в системе реализованы аппаратные прерывания, то есть прерывания от внешних устройств.

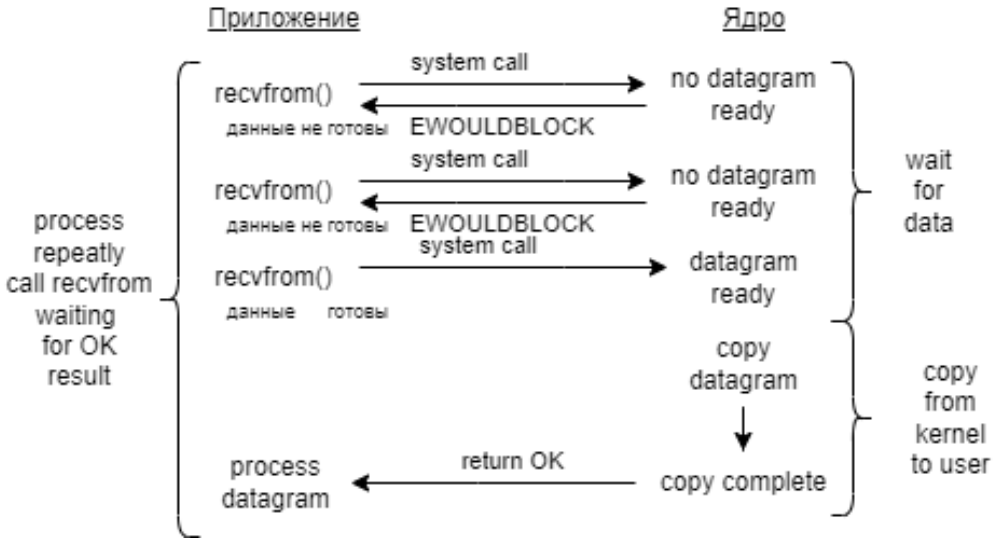
В современных системах для отдельно стоящей машины блокирующий ввод-вывод – основная модель ввода-вывода (запросив ввод-вывод, приложение блокируется, система переходит в режим ядра и запрос обслуживается).

При этом в блокирующем вводе-выводе в полной мере реализована идея распараллеливания функций (3-е поколение ЭВМ), которая базируется на стремлении освободить процессор от непроизводительных действий (управляется медленными внешними устройствами).

Пока медленное внешнее устройство выполняет задачу ввода данных, процессор может перейти на выполнение другой работы.

1.1.2. Неблокирующий ввод-вывод

Исторически был реализован раньше блокирующего. Управление операцией ввода-вывода осуществляет процессор: он опрашивал флаг устройства, который информировал процессор о готовности данных.



Опрос (polling) – периодическое действие: опрос выполняется до тех пор, пока данные не будут готовы, то есть пока они не поступили в буфер ядра и из него уже в буфер приложения.

При этом процесс не блокируется, но процессорное время тратиться на опрос.

E-Error: ошибка обращения, данные не готовы.

Для этого определяется специальные возвращаемые значения для каждой конкретной модели ввода-вывода. Это учитывается разработчиками системы.

Здесь нет аналогии с активным ожиданием на процессоре: это концептуально разные вещи. В данном случае идет опрос флага о готовности контроллера прерывания, то есть это обращение к внешнему устройству.

В система должны быть специальные команды для работы в режиме polling, однако это устаревший способ взаимодействовать с внешними устройствами. Он затраный, но исключить опрос для любой системы мы не можем, В каких-то системах допускается возможность реализации такого опроса, а в наших – нет, так как у нас есть прерывания от внешних устройств (блокрующий ввод-вывод).

1.1.3. Мультиплексирование ввода-вывода

Мультиплексирование всегда рассматривается для модели «клиент-сервер» на соекетах (абстракция конечной точки соединения, средство взаимодействия параллельных процессов в BSD, универсальны для отдельно стоящих машин и для распределенных систем).

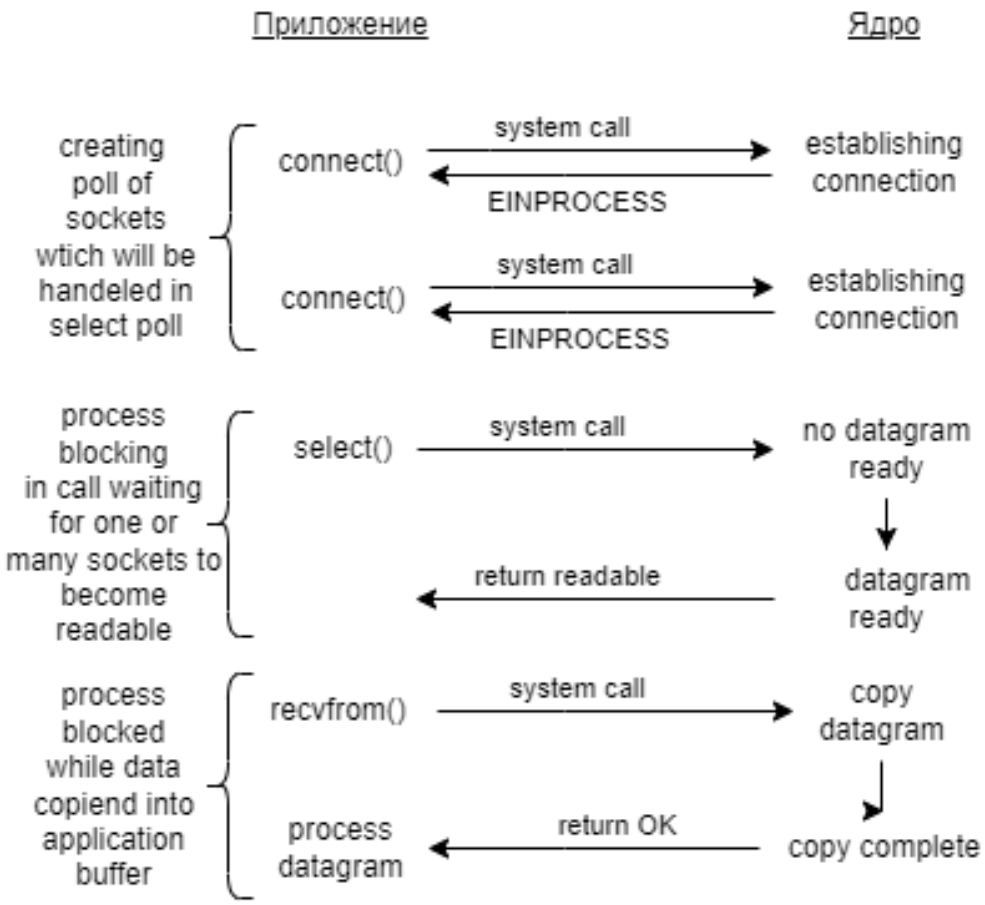
Мультиплексирование (асинхронный в/в) – очень хорошая альтернатива многопоточности. Многопоточность очень затратная, особенно на UNIX/Linux (а Windows заточен на потоки, а не на процессы).

Для реализации мультиплексирования система предоставляет специальные системные вызовы (мультиплексеры) select и poll.

select() == pselect()

poll() == epoll() в современных системах.

Сокеты – тоже альтернатива многопоточности.



Формируется пул сокетов, которые будут обрабатываться в петле select.

На select сервер будет блокирован, но время блокировки будет меньше, так как опрашивается сразу пул сокетов. Вероятность готовности какого-то либо сокета из множества выше вероятности готовности конкретного сокета.

Взаимодействие так же выполняется путем передачи сообщений, но receive from выкинули, иначе диаграмма была бы слишком большой. Поэтому указывается только select.

Мультиплексирование используется только для сетей (распределенных систем).

Мультиплексирование позволяет сократить время блокировки (блокировки – зло, но зло неизбежное) за счет того, что мультиплексор фактически опрашивает готовность точек соединения (на своей стороне, так как клиенты устанавливают соединение с сервером).

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

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

После того, как select определил, что соединение установлено, вызывается receive from.

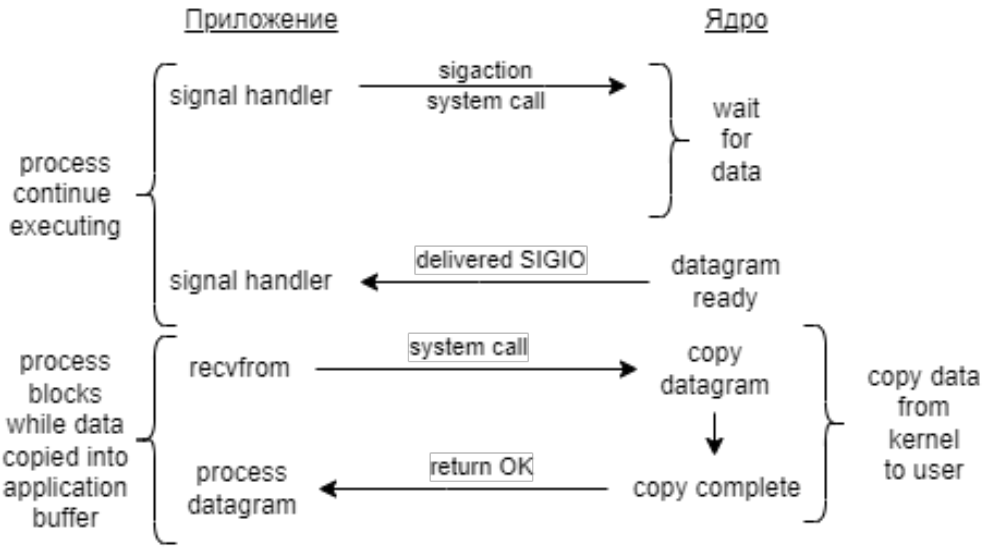
Когда данные готовы, возвращается readable (данные доступны), вызывается receive from и производится копирование из буфера ядра в буфер приложения.

process datagram – обработка данных.

select вызывается на стороне сервера, сервер однопоточный, так как мультиплексирование – альтернатива многопоточности.

Пример мультиплексора – переключатель режимов работы стиральной машины.

1.1.4. Ввод-вывод, управляемый сигналом



Устанавливается обработчик специального сигнала SIGIO, задача которого – проинформировать процесс о том, что данные готовы (delivered SIGIO – доставка сигнала процессу).

Процесс вызывает receive from, чтобы получить данные.

Буфер устройства → Буфер ядра → Буфер приложения.

Процесс продолжает выполняться и блокируется только на время копирования данных, то есть время блокировки уменьшилось.

Всю работу в данном способе ввода-вывода берет на себя ядро: оно отслеживает готовность данных и после этого посылает сигнал SIGIO. В результате будет вызван обработчик этого сигнала.

Это (обработчик) называется callback-функцией (функцией обратного вызова).

При этом receive from можно выполнить либо в обработчике сигнала, либо в главном потоке программы.

Сигнал типа SIGIO для каждого процесса может быть только один.

В результате в каждый момент времени можно работать только с одним файловым дескриптором.

На время выполнения обработчика сигнала данный сигнал блокируется.

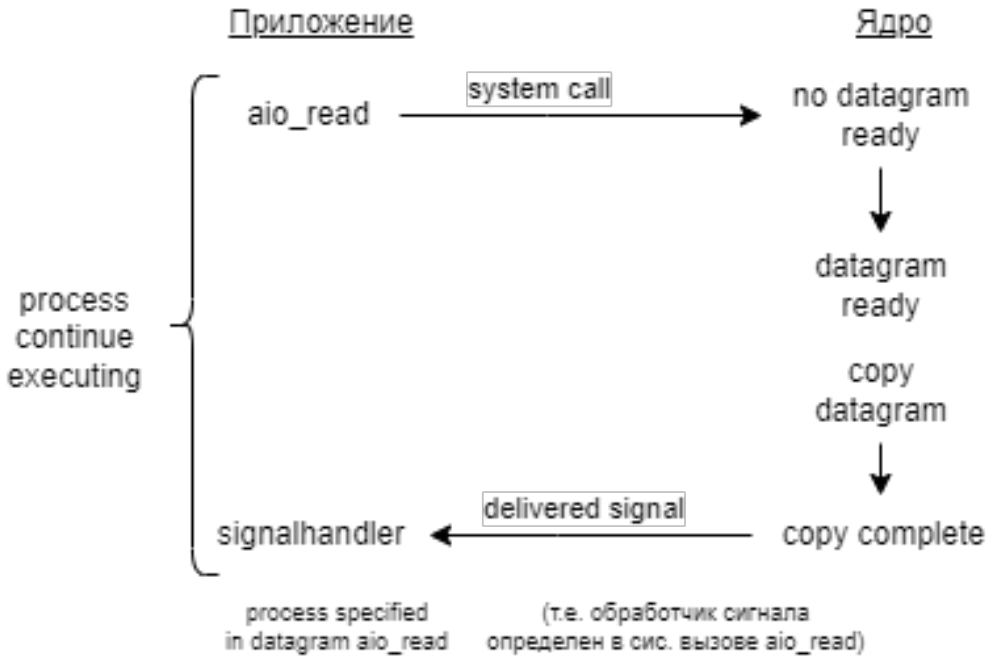
Если в период блокировки доставляются несколько сигналов, то они теряются.

Если маска сигнала (sa_mask) установлена в NULL, то на время выполнения обработчика сигнала другие сигналы не блокируются.

1.1.5. Асинхронный ввод-вывод

Блокировки увеличивают время выполнения приложения, время блокировки – случайная величина. Разработчики стремятся избежать блокировок и системы предоставляют соответствующие системные вызовы, обеспечивающие асинхронный ввод-вывод.

Для его реализации система предоставляет пользователю специальные системные вызовы. Как правило, они начинаются с a – asynchronous.



Процесс не блокируется и продолжает выполняться. В отличие от предыдущей модели, сигнал информирует процесс о полном завершении операции ввода-вывода, включая копирование данных.

Здесь вместо receive from используется read/write, так как здесь нет передачи сообщений, это в чистом виде ввод-вывод на отдельно стоящей машине.

receive from/send to – системные вызовы, связанные с приемом/посылкой сообщений. read/write – системные вызовы, связанные с получением данных от внешних устройств (по

факту из файла, так как в UNIX все файл).

Асинхронный ввод-вывод определен в POSIX (спецификация POSIX, согласовавшая все различия в функциях осинхронного ввода-вывода, возникших в разных системах, объединив их достоинства.

Все функции асинхронного ввода/вывода работают таким образом, что они сообщают ядру о начале операции ввода-вывода, а процесс они уведомляют о завершении операции ввода-вывода, включая копирование данных.

Основное отличие асинхронного ввода-вывода от ввода-вывода, управляемого сигналом: при вводе-выводе, управляемым сигналом, сигнал информирует процесс и готовности данных, а при асинхронном вводе-выводе процесс вообще не блокируется и информируется о полном завершении операции ввода-вывода.

Проблема асинхронного ввода-вывода: необходимо получать асинхронные события синхронно, так как данные нужны приложению для выполнения дальнейших действий.

В асинхронном вводе-выводе внимание сосредотачивается на 2 моментах:

1. На возможности определить, что ввод-вывод можно выполнить быстро (в технике нет быстро/медленно);

2. На завершении операции ввода-вывода в случае невозможности немедленного выполнения ввода-вывода, то есть возвращения ошибки.

1.1.6. Классификация моделей ввода-вывода

S — Synchronous A — Asynchonous

	Blocking	Non-blocking
S	1) recv from / send to /read /write - в разных источниках по-разному	2) polling (опрос)
A	3) IO multiplexing 4) SIGIO	5) AIO

1.2. Сетевой стек

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

В сетях - только передача сообщений, которые должны сопровождаться адресом

Пакет - сообщение с адресом + служебная информация

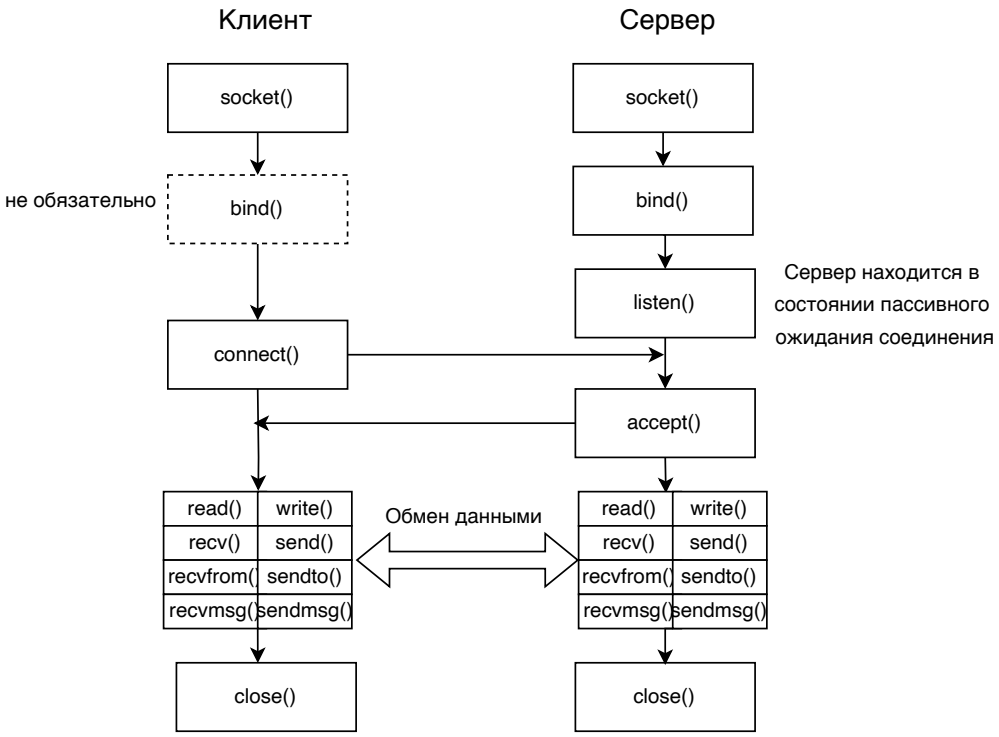
В Linux определен интерфейс между пользовательскими процессами и стеком сетевых протоколов в ядре.

Это не по семинару*

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

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

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



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

AF_INET, SOCK_STREAM - сетевое взаимодействие по протоколу TCP

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

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

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

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

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

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

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

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

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

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

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

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

> про уровни сетевых протоколов
>
> Протоколы различаются по уровням. Нижний уровень - непосредсвенное взаимодействие с аппаратной частью (самое важное)

1.3. Адресация сокетов и ее особенности для разных типов сокетов

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

Взаимодействие на сокетах происходит по модели клиент-сервер

Адресация сокетов:

```
1 struct sockaddr
2 {
```

```
3   sa_family_t sa_family;
4   char sa_data[14];
5 }
```

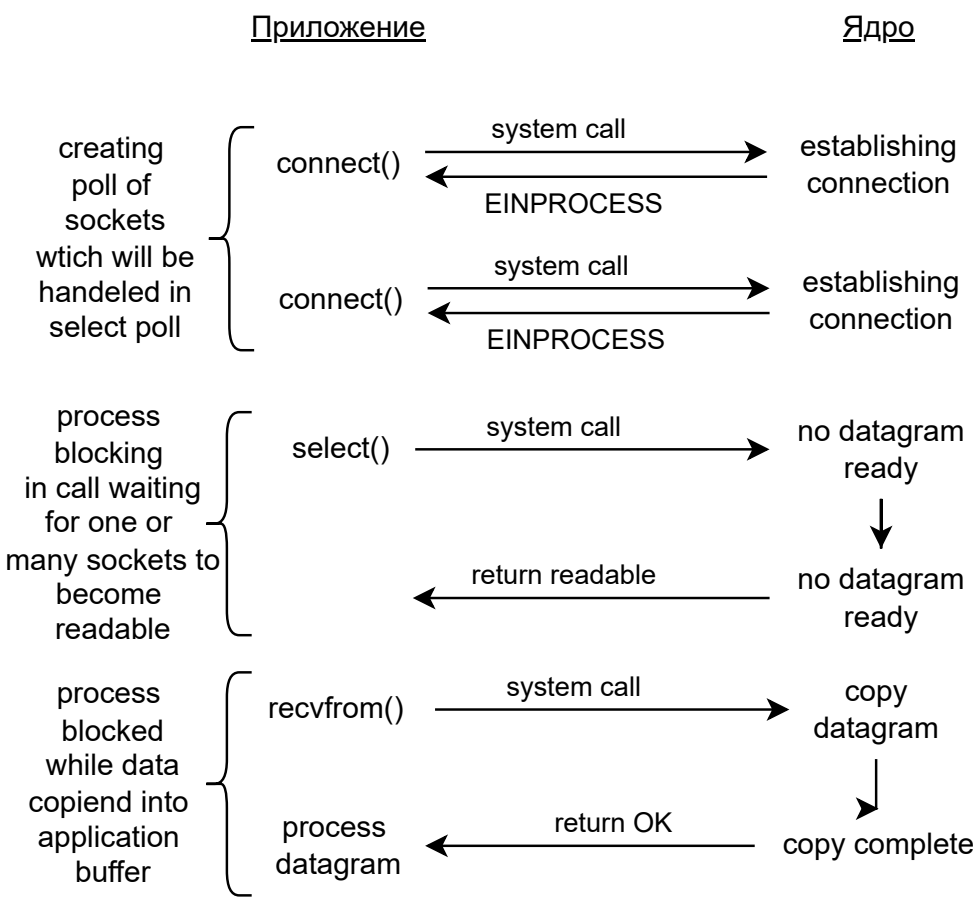
Такая структура адреса не подходит для интернета, так как там необходимо указывать номер порта и сетевой адрес. Для интернета разработана другая структура:

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

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

Сетевые сокеты с мультиплексированием:

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



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

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

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

Для создания пула соединений можно использовать массив.

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

Мультиплексопы: select poll pselect epoll

AF_INET — сокеты семейства протоколов TCP/IP для интернета версии 4 (Inet4/IPv4) (интернет домен) любая компьютерная сеть

SOCK_STREAM — сокеты потоков. Ориентированы на потоки, надежно упорядоченная полнодуплексное логическое соединение между двумя сокетами (TCP).

Код клиента

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netinet/in.h>
6  #include <netdb.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <fcntl.h>
10 #include <errno.h>
11
12 #define SERVER_PORT 8080
13 #define MSG_LEN 64
14
15 int main(void)
16 {
17    setbuf(stdout, NULL);
18
19    struct sockaddr_in serv_addr =
20    {
21       .sin_family = AF_INET,
22       .sin_addr.s_addr = INADDR_ANY,
23       .sin_port = htons(SERVER_PORT)
24    };
25    socklen_t serv_len;
26
27    char buf[MSG_LEN];
28
29    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
30    // error handling
31
32    if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
          0)
```

```
33      // error handling
34
35    char input_msg[MSG_LEN], output_msg[MSG_LEN];
36    sprintf(output_msg, "%d", getpid());
37
38    if (write(sock_fd, output_msg, strlen(output_msg) + 1) == -1)
39      // error handling
40
41    if (read(sock_fd, input_msg, MSG_LEN) == -1)
42      // error handling
43
44    printf("Client receive: %s \n", input_msg);
45    close(sock_fd);
46    return EXIT_SUCCESS;
47  }
```

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

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

struct sockaddr — структура адреса.

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

EPOLLIN — событие активно, когда есть данные на вход.

EPOLLOUT — событие активно, когда есть данные на выход.

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

По сути, без этого флага разблокировка epoll будет осуществляться каждый раз, когда есть необработанное событие, то есть если в сокете появились данные, доступные для чтения, и не было произведено чтения, то при следующем вызове epoll_wait снова произойдет

разблокировка. В случае установки

EPOLLET разблокировка при повторном epoll_wait не произойдет, даже если данные не были прочитаны на прошлой итерации. Флаг EPOLLET это то, что делает epoll O(1) мультиплексором для событий — очень быстро.

setsockopt(sock, SOL_SOCKET,

SO_REUSEADDR, &sopt, sizeof(sopt) — установка опций сокета: SOL_SOCKET — нужен для манипуляции флагами сокета, SO_REUSEADDR — повторное использование адреса после вызова accept. sopt — заполняется.

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

epoll_wait(epoll_fd,&epev,

CLIENTS_MAX+1, 1) — опрос.

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

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

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

Код сервера

```
1  #include <arpa/inet.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/epoll.h>
5  #include <sys/socket.h>
6  #include <unistd.h>
7
8  #define CLIENTS_MAX 5
9  #define PORT 8181
10 #define BUF_SIZE 128
11
12 int main()
13 {
14   int epoll_fd;
15
16   if ((epoll_fd = epoll_create(CLIENTS_MAX)) == -1)
```

```
17  {
18    perror("Can't epoll_create");
19    exit(1);
20  }
21
22  int sock;
23
24  if ((sock = socket(AF_INET, SOCK_STREAM | O_NONBLOCK, 0)) == -1)
25  {
26    perror("Can't socket");
27    exit(1);
28  }
29
30  int sopt = 1;
31
32  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sopt, sizeof(sopt)) ==
        -1)
33  {
34    perror("Can't setsockopt");
35    exit(1);
36  }
37
38  struct sockaddr_in addr;
39
40  addr.sin_family = AF_INET;
41  addr.sin_addr.s_addr = INADDR_ANY;
42  addr.sin_port = htons(PORT);
43
44  if (bind(sock, (struct sockaddr *) &addr, sizeof(addr)) == -1)
45  {
46    perror("Can't bind");
47    exit(1);
48  }
49
50  if (listen(sock, CLIENTS_MAX) == -1)
51  {
52    perror("Can't listen");
53    exit(1);
```

```
54    }
55
56    struct epoll_event epev;
57
58    epev.events = EPOLLIN;
59
60
61    epev.data.fd = sock;
62
63    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sock, &epev) == -1)
64    {
65      perror("Can't epoll_ctl");
66      exit(1);
67    }
68
69    while (1)
70    {
71      struct epoll_event epev[CLIENTS_MAX + 1];
72      int num;
73
74      if ((num = epoll_wait(epoll_fd, &epev, CLIENTS_MAX + 1, -1)) == -1)
75      {
76        perror("Can't epoll_wait");
77        exit(1);
78      }
79
80      for (int i = 0; i < num; i++)
81      {
82        if (epev[i].data.fd == sock)
83        {
84          int conn;
85
86          if ((conn = accept(sock, NULL, NULL)) == -1)
87          {
88            perror("Can't accept");
89            exit(1);
90          }
91
```

```
92          struct epoll_event epev;
93
94          int flags = fcntl(conn, F_GETFL, 0);
95          fcntl(conn, F_SETFL, flags | O_NONBLOCK);
96
97          epev.events = EPOLLIN | EPOLLET ;
98          epev.data.fd = conn;
99
100         if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn, &epev) == -1)
101         {
102           perror("Can't epoll_ctl");
103           exit(1);
104         }
105       }
106
107       else
108       {
109         int conn = epev[i].data.fd;
110
111         char received_msg[BUF_SIZE], send_msg[BUF_SIZE];
112
113         if (recv(conn, received_msg, sizeof(received_msg), 0) == -1)
114         {
115           perror("Can't recv");
116           exit(1);
117         }
118
119         printf("Server %d received message: %s\n", getpid(), received_msg)
              ;
120         sprintf(send_msg, "%s from server with pid %d", received_msg,
              getpid());
121
122         if (send(conn, send_msg, sizeof(send_msg), 0) == -1)
123         {
124           perror("Can't send");
125           exit(1);
126         }
127
```

```
128             printf("Server %d send message: %s\n", getpid(), send_msg);
129             close(conn);
130         }
131     }
132   }
133
134   return 0;
135 }
```