

Сетевое взаимодействие в UNIX




Сокеты (гнезда)

- Универсальный механизм межпроцессного взаимодействия
- Используется для локального взаимодействия (аналогично именованным каналам)
- Используется для сетевого взаимодействия




Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Возвращается файловый дескриптор сокета
 - В зависимости от типа создаваемого сокета может потребоваться дополнительная настройка параметров
 - Файловый дескриптор может копироваться с помощью `dup`, наследоваться через `fork/exec`
 - В конце использования файловый дескриптор должен быть закрыт с помощью `close`
- 

Параметры создания сокета

- Параметр `domain` — домен сокета
 - `PF_UNIX, PF_LOCAL` — локальный сокет
 - `PF_INET` — IPv4
 - `PF_INET6` — IPv6
 - Параметр `type` — тип соединения
 - `SOCK_STREAM` — потоковый сокет
 - `SOCK_DGRAM` — датаграммный сокет
 - Параметр `protocol` уточняет используемый протокол для пары `(domain,type)`. 0 — выбрать протокол по умолчанию
- 

Примеры создания сокета

```
fd = socket(PF_LOCAL, SOCK_STREAM, 0);
```

- Локальный потоковый сокет (аналог именованных каналов)

```
fd = socket(PF_INET, SOCK_STREAM, 0);
```

- Сокет для работы по протоколу TCP

```
fd = socket(PF_INET, SOCK_DGRAM, 0);
```

- Сокет для работы по протоколу UDP

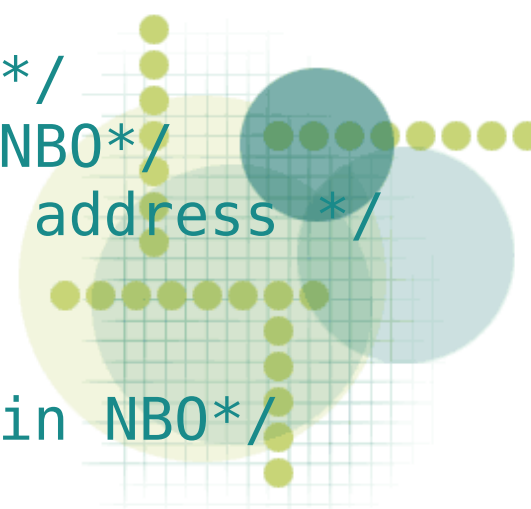


Адрес соединения

- Для указания адреса как отправителя, так и получателя используется `struct sockaddr`
- При работе с каждым конкретным доменом сокета должна использоваться структура адреса, специфичная для этого домена


```
struct sockaddr_in {
    sa_family_t    sin_family; /* AF_INET */
    uint16_t       sin_port;   /* port in NBO*/
    struct in_addr sin_addr;   /* internet address */
};

struct in_addr {
    uint32_t       s_addr;     /* address in NBO*/
};
```



Получение информации об адресе

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node` — строка имени или IP-адреса хоста
 - `service` — строка номера порта или имени сервиса
 - `hints` — флаги для управления трансляцией
 - `res` — адрес указателя, в который помещается указатель на голову списка результатов трансляции
- 

Структура информации об адресе

```
struct addrinfo {
    int          ai_flags;        // флаги работы
    int          ai_family;      // IPv4 или IPv6
    int          ai_socktype;    // STREAM или DGRAM
    int          ai_protocol;    // протокол
    size_t      ai_addrlen;
    struct sockaddr *ai_addr;
    char        *ai_canonname;
    struct addrinfo *ai_next;    // след. эл. списка
};
```

- Флаги: AI_NUMERICHOST, AI_CANONNAME, AI_PASSIVE, AI_NUMERICSERV



Пример

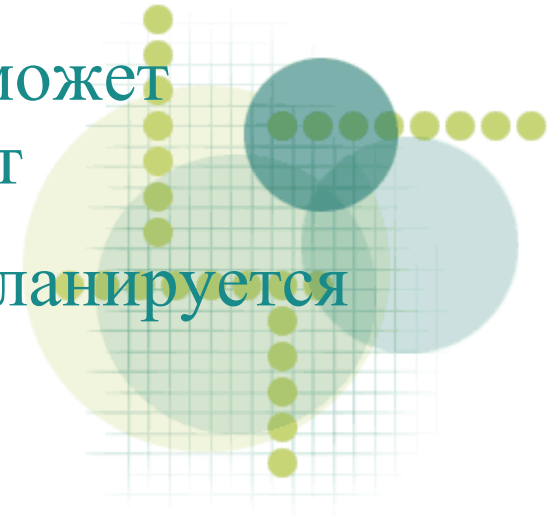
```
int main(int argc, char *argv[]) {
    struct addrinfo hints, *reslist = NULL;
    memset(&hints, 0, sizeof(hints));
    hints.ai_flags = AI_NUMERICSERV;
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    int res = getaddrinfo(argv[1], argv[2], &hints,
                          &reslist);

    if (res != 0) {
        fprintf(stderr, "Error: %s\n",
                gai_strerror(res));
        exit(1);
    }
    // ...
    freeaddrinfo(res);
    return 0;
}
```



Исходящий порт

- Для полной идентификации датаграммы или соединения необходим номер исходящего порта
- Если номер исходящего порта не задан, он назначается автоматически
- Для взаимодействия по протоколу TCP исходящий порт, как правило, не требуется
- Для взаимодействия по протоколу UDP может потребоваться назначить исходящий порт
- Привязку необходимо выполнять, если планируется принимать сообщения



Привязка порта

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

- `addr` — структура, задающая параметры привязки
- `addrlen` — размер структуры адреса



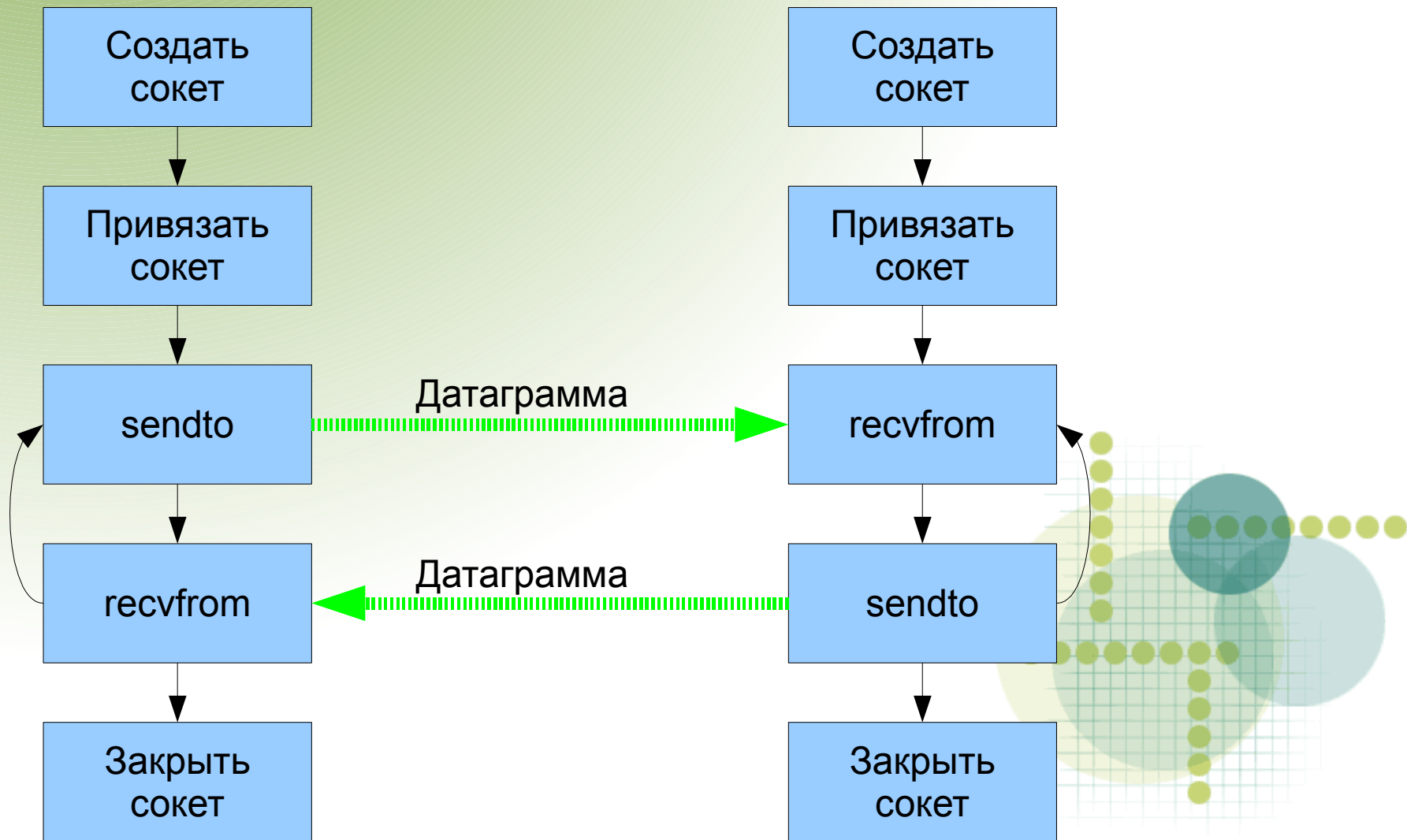
Пример

```
struct sockaddr_in sa;  
memset(&sa, 0, sizeof(sa));  
sa.sin_family = AF_INET;  
sa.sin_port = htons(23362);  
sa.sin_addr.s_addr = INADDR_ANY;  
bind(sockfd, (struct sockaddr*) &sa, sizeof(sa));
```

- В данном примере выполняется привязка к номеру порта 23362
- INADDR_ANY указывает, что можно взять любой подходящий IP-адрес данного хоста



Взаимодействие без установления соединения



Прием/передача данных

```
ssize_t sendto(int s, const void *buf, size_t len,  
               int flags, const struct sockaddr *to,  
               socklen_t tolen);  
ssize_t recvfrom(int s, void *buf, size_t len,  
                 int flags, struct sockaddr *from,  
                 socklen_t *fromlen);
```

- `flags` позволяет задавать флаги отправки/приема:
 - `MSG_DONTWAIT` — неблокирующий прием
 - `MSG_PEEK` — не удалять данные из буфера приема
 - `MSG_TRUNC` — обрезать входной пакет



Отправка данных на адрес, заданный в ком. строке

```
// reslist – указатель на список структур адреса
// предположим, что создание сокета и
// bind уже выполнены
// в buf сформирована датаграмма размера len
int res = sendto(sockfd, buf, len, 0,
                reslist->ai_addr,
                reslist->ai_addrlen);

if (res != len) {
    // обработать ошибочную ситуацию
    // errno == EMSGSIZE – датаграмма велика
}
```



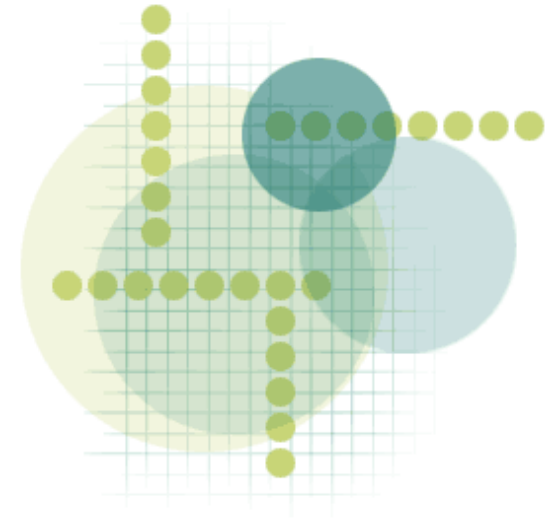
Получение данных

```
// предположим, что создание сокета и
// bind уже выполнены
// buf – буфер для приема размера len
struct sockaddr_in inaddr;
socklen_t inlen = sizeof(inaddr);
memset(&inaddr, 0, sizeof(inaddr));
int rsz = recvfrom(sockfd, buf, len, MSG_TRUNC,
                  (struct sockaddr*) &inaddr,
                  &inlen);
printf("Message from %s, port %d\n",
       inet_ntoa(inaddr.sin_addr),
       ntohs(inaddr.sin_port));
```



Отправка ответа отправителю

```
// предположим, что создание сокета и  
// replybuf – датаграмма для отправки  
// replylen – ее размер  
int wsz = sendto(sockfd, replybuf, replylen, 0,  
                 (struct sockaddr*) &inaddr,  
                 inlen);
```

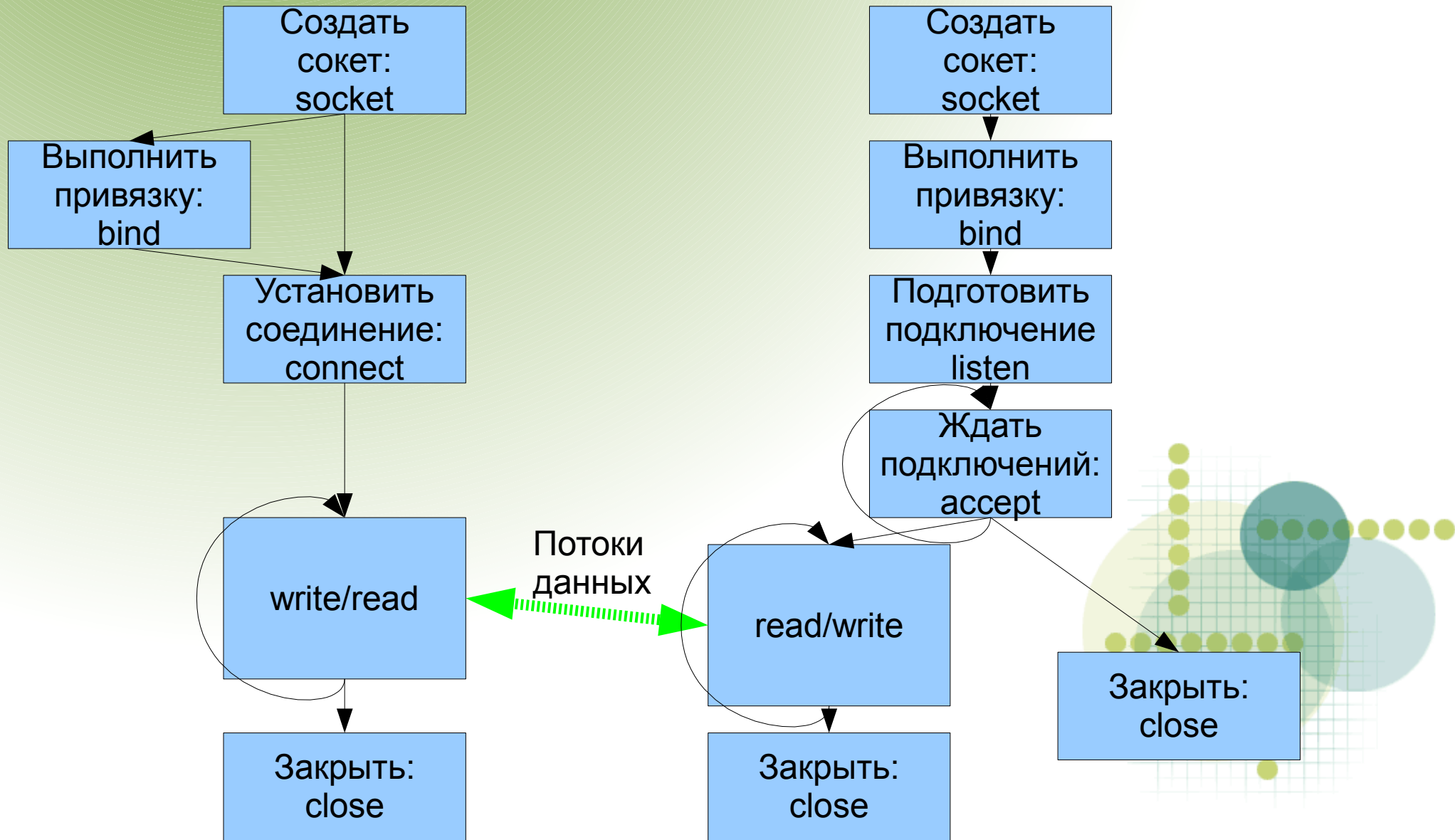


Архитектура клиент-сервер

- Техническое понимание:
 - Клиент — сторона, которая инициирует соединение
 - Сервер — сторона, которая ожидает подключения
- Логическое понимание
 - Клиент — сторона, запрашивающая выполнение некоторого сервиса
 - Сервер — сторона, предоставляющая сервис
- Как правило, техническое = логическое



Взаимодействие с установлением соединения



Подключение к серверу

```
int connect(int sockfd, const struct sockaddr *sa,  
            socklen_t addrlen);
```

- Пример: подключение к хосту/порту, указанным в командной строке

```
int res = connect(sockfd, reslist->ai_addr,  
                  reslist->ai_addrlen);  
if (res < 0) {  
    // обработать ошибочную ситуацию  
}
```



Переключение сокета в режим прослушивания

```
int listen(int sockfd, int backlog);
```

- Параметр `backlog` — длина очереди запросов, ожидающих обработки, в ядре
- Если программа не успевает обрабатывать входящие запросы на подключение, ядро будет отвергать запросы, которые бы приводили к превышению длины очереди значения `backlog`
- Обычное «магическое» значение — 5



Ожидание подключения

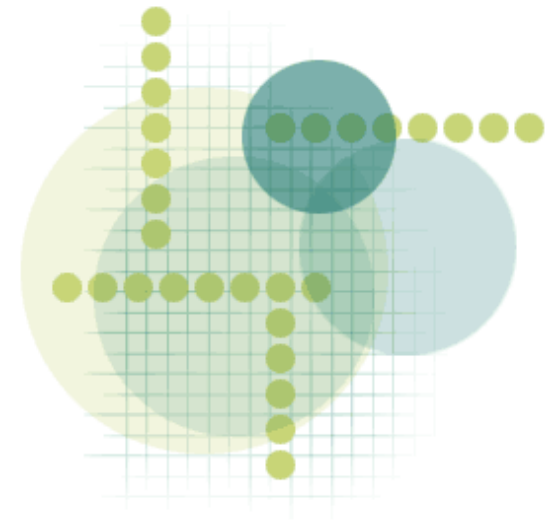
```
int accept(int sockfd, struct sockaddr *addr,  
          socklen_t *addrlen);
```

- В структуру `addr` возвращается адрес подключившегося клиента
- При успехе `accept` возвращает *новый* файловый дескриптор, который используется для обмена данными с клиентом
- Файловый дескриптор `sockfd` можно использовать для подключения других клиентов



Ожидание подключения

```
struct sockaddr_in inaddr;  
socklen_t inlen = sizeof(inaddr);  
memset(&inaddr, 0, sizeof(inaddr));  
int afd = accept(sockfd,  
                (struct sockaddr*) &inaddr,  
                &inlen);  
  
// при успехе ф. д. afd используется для обмена  
// данными
```



Обработка входящих подключений

- При каждом успешном выполнении ассерт создается новый файловый дескриптор для обмена данными с клиентами
- Сервер должен выполнять операции ввода/вывода с несколькими файловыми дескрипторами и обрабатывать новые подключения
- Каждая операция может заблокировать процесс на неопределенное время



Архитектура сервера

- Один процесс на клиента
- Одна нить на клиента
- Мультиплексирование событий ввода-вывода



Один процесс на клиента

- Для взаимодействия с клиентом создается отдельный процесс
- Основной процесс продолжает обрабатывать входящие подключения и отслеживать завершение порожденных процессов
- Можно применять: `httpd`, `ftpd`, `sshd`
- Недостатки:
 - Сложность организации взаимодействия клиентов
 - Плохая масштабируемость



Один процесс на клиента

- Для отслеживания завершения процессов можно установить обработчик сигнала SIGCHLD в режиме отключенного перезапуска системных вызовов (с помощью sigaction)
- Обработчик может ничего не делать
- В этом случае ассерт будет прерываться при приходе сигнала



Один процесс на клиента

```
while (1) {
    struct sockaddr_in inaddr;
    socklen_t inlen = sizeof(inaddr);
    memset(&inaddr, 0, sizeof(inaddr));
    int afd = accept(sockfd, (struct sockaddr*) &inaddr,
                    &inlen);
    if (afd < 0 && errno == EINTR) {
        while ((pid=waitpid(-1, &status, WNOHANG)) > 0) {
            // обработать завершения порожденного процесса
        }
    }
    if (!fork()) {
        close(sockfd);
        // работать с клиентом:
        // read(afd, ...); write(afd, ...);
        _exit(0);
    }
}
```



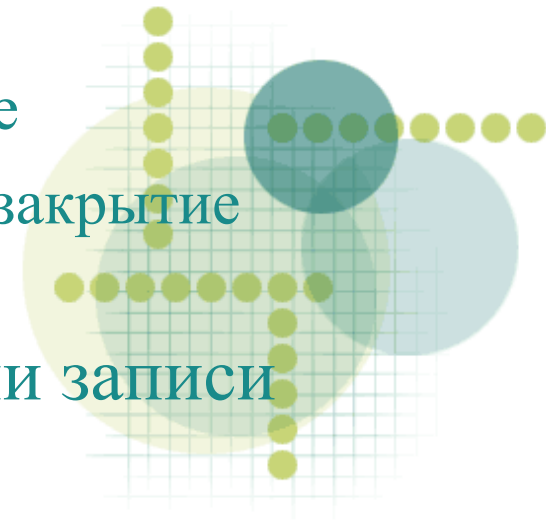
Одна нить на клиента

- Для взаимодействия с клиентом создается отдельная нить
- Основная нить продолжает обрабатывать входящие подключения
- Взаимодействие клиентов друг с другом организуется через структуры данных в общем адресном пространстве
- Недостатки:
 - Необходимость корректного доступа к разделяемым ресурсам
 - Недостаточная масштабируемость



Мультиплексирование событий

- Процесс-сервер отслеживает события на файловых дескрипторах и приход сигнала
- Типы возможных событий:
 - Ф. д. готов к выполнению операции чтения, т. е. операция чтения не заблокирует процесс
 - Поступили данные
 - Поступил запрос на подключение
 - Поступил признак конца файла (закрытие соединения)
 - Ф. д. готов к выполнению операции записи
 - Поступил сигнал



Множества ф. д.

```
#include <sys/select.h>
```

```
void FD_CLR(int fd, fd_set *set);  
int  FD_ISSET(int fd, fd_set *set);  
void FD_SET(int fd, fd_set *set);  
void FD_ZERO(fd_set *set);
```

- Функции работают с множеством файловых дескрипторов
- Максимальный номер файлового дескриптора в множестве зависит от ОС (обычно 1024)

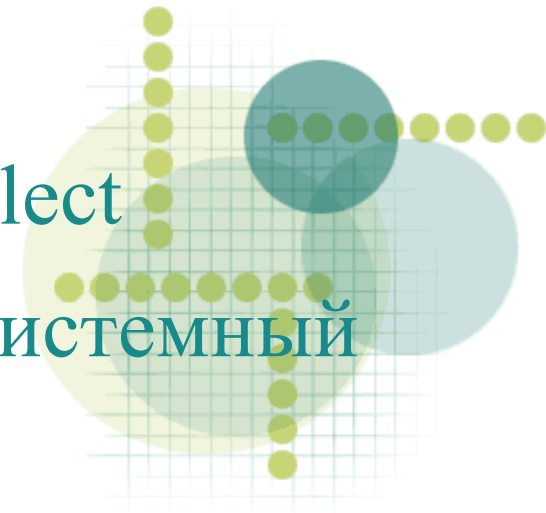


СИСТЕМНЫЙ ВЫЗОВ pselect

```
#include <sys/select.h>
```

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

- В более старых системах может использоваться системный вызов `select`
- Кроме того может использоваться системный вызов `poll`



СИСТЕМНЫЙ ВЫЗОВ pselect

- `readfds` — множество файловых дескрипторов, проверяемых на готовность к операции чтения (допускается `NULL`)
- `writefds` — множество файловых дескрипторов, проверяемых на готовность к операции записи (допускается `NULL`)
- `exceptfds` — множество файловых дескрипторов, проверяемых на «срочные» данные (обычно `NULL`)
- `nfds` — максимальный номер файлового дескриптора во всех трех множествах + 1



СИСТЕМНЫЙ ВЫЗОВ pselect

- `timeout` задает максимальное время ожидания, `NULL` — неограниченное ожидание

```
struct timespec {  
    long    tv_sec;           /* seconds */  
    long    tv_nsec;        /* nanoseconds */  
};
```

- `sigmask` задает маску блокируемых сигналов на время работы `pselect` (по аналогии с `sigsuspend`)



СИСТЕМНЫЙ ВЫЗОВ pselect

- pselect возвращает
 - -1 при ошибке, например, при поступлении и обработке сигнала
 - 0 при истечении времени ожидания
 - >0 — суммарное число файловых дескрипторов, ГОТОВЫХ К ВЫПОЛНЕНИЮ операции



Использование pselect

- Для каждого клиента хранится информация:
 - Номер файлового дескриптора
 - Состояние обработки данных
 - Буфер данных, ожидающих отправки клиенту



Схема использования pselect

```
while (1) {  
    FD_ZERO(&rds); FD_ZERO(&wds);  
    // добавить в rds ф. д. сокета для подключений  
    // добавить в rds ф. д. клиентов, от которых  
    // ожидается получение данных  
    // добавить в wds ф. д. клиентов с непустым  
    // буфером данных на отправку  
    // вычислить макс. номер ф. д. +1  
    // установить желаемое значение timeout  
    // установить множество блокируемых сигналов  
    // на время ожидания  
    res = pselect(nfd, &rds, &wds, NULL,  
                  &timeout, &sigmask);
```

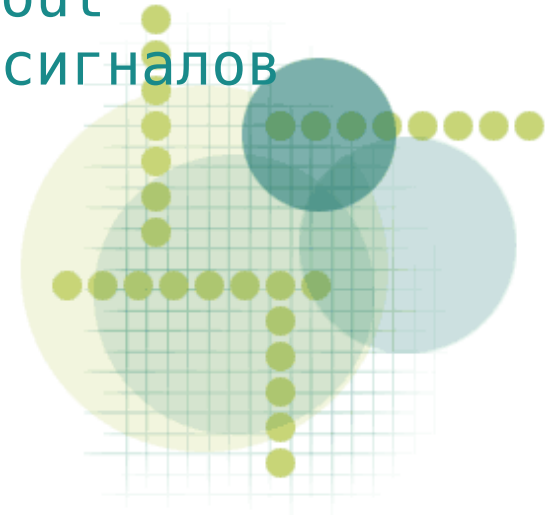


Схема использования pselect

```
if (res < 0 && errno == EINTR) {  
    // обработать поступление сигнала  
    continue;  
}  
if (res < 0) {  
    // обработать прочие ошибки  
    continue;  
}  
if (res == 0) {  
    // обработать тайм-аут  
    continue;  
}  
if (FD_ISSET(sockfd, &rds)) {  
    // обработать подключение нового клиента  
}
```



Схема использования pselect

```
// для каждого клиента:  
// если готов ф. д. чтения данных, считать  
// очередную порцию данных, если входной  
// запрос полностью сформирован,  
// обработать его  
// если готов ф. д. записи данных,  
// записать очередную порцию данных
```



Событийно-ориентированные программы

- Программа, использующая `pselect` схематично выглядит так:

```
while (1) {  
    // ждать поступления события  
    // обработать поступившее событие  
}
```

- Единственной точкой ожидания процесса является `pselect`
- Это — пример событийно-ориентированной программы



Событийно-ориентированные программы

- Событийно-ориентированные программы построены по принципу автоматов:
 - Выделяются состояния, в которых может находиться автомат
 - Выделяются все типы событий
 - Описываются переходы между состояниями по приходу всех типов событий
 - Требования: действия во время выполнения переходов между состояниями не должны блокировать процесс

