

ОСНОВЫ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

Сигналы и средства межпроцессного взаимодействия

Гирик Алексей Валерьевич

Университет ИТМО
2021

Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
 - shorturl.at/gjABL



Средства межпроцессного взаимодействия

- Сигналы
 - реального времени
- Каналы
 - безымянные каналы
 - именованные каналы
- XSI IPC
 - Очереди сообщений
 - Семафоры
 - Разделяемая память
- Сокеты
 - локальные сокеты
 - сетевые сокеты

локальные IPC
(в пределах хоста)

Сигналы

Сигналы

- Сигнал – это асинхронное уведомление процесса о наступлении некоторого события
- Источником сигнала могут быть:
 - ядро ОС
 - другой процесс
 - терминал

Действие сигнала

- Или, в случае некоторых сигналов, **воздействие**
- Действие по умолчанию
 - завершение процесса
 - завершение с созданием файла core
 - остановка процесса
 - игнорирование
- Обработка
 - вызов установленного обработчика сигнала

Особенности обработки сигналов

- для большинства сигналов действие по умолчанию – завершение процесса
[https://ru.wikipedia.org/wiki/Сигнал_\(Unix\)](https://ru.wikipedia.org/wiki/Сигнал_(Unix))
- не все сигналы могут быть обработаны (перехвачены) – SIGKILL и SIGSTOP могут быть перехвачены только процессом init (с PID == 1)

Список сигналов и отправка сигналов из КОНСОЛИ

- `kill -l`
 - отображает список доступных в системе сигналов
- `kill -N pid`
- `kill -NAME pid`
 - отправляет сигнал процессу с заданным pid

Особенности обработки сигналов

- не все сигналы представлены на всех POSIX-системах; стандартными являются сигналы с номерами 1 .. 28

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

```
#ifdef SIGPWR
```

```
...
```

```
#endif
```

Переносимая отправка сигналов из программы

- Функция стандартной библиотеки `raise()` позволяет отправить сигнал самой себе

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

```
int raise(int sig);
```

Отправка сигнала заданному процессу

- Системный вызов `kill()` позволяет отправить сигнал заданному процессу:

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

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

```
int kill(pid_t pid, int sig);
```

Переносимая обработка сигналов в программе

- Функция стандартной библиотеки `signal()`:

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

```
void (*signal(int signum, void (*handler)(int)))(int);
```

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

Возможные проблемы при обработке сигналов

- сбрасывание обработчика сигналов после первого вызова в некоторых POSIX-системах (не в Linux)
 - состояние гонки, которое возникает в результате сброса обработчика – можно пропустить сигналы и тогда будет вызван обработчик по умолчанию

Возможные проблемы при обработке сигналов

- одновременное поступление нескольких разных сигналов – возможно непредсказуемое поведение, если обработчики сигналов "мешают" друг другу

```
volatile sig_atomic_t SIGINT_flag = 0;  
volatile sig_atomic_t SIGQUIT_flag = 0;
```

```
void SIGINT_handler(int signum) {  
    SIGINT_flag = 1;  
}
```

```
void SIGQUIT_handler(int signum) {  
    SIGQUIT_flag = 1;  
}
```

Возможные проблемы при обработке СИГНАЛОВ

- внутри обработчика можно вызывать только "безопасные" (*async-signal-safe*) функции – большая часть функций стандартной библиотеки к ним не ОТНОСИТСЯ
 - https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04
- `man 7 signal-safety`

Возможные проблемы при обработке сигналов

- системные вызовы, которые находились в процессе выполнения, могут вернуться с ошибкой EINTR, после чего могут быть перезапущены (а могут и не быть перезапущены)

```
while ((n = read(fd, buf, sizeof(buf) > 0) {  
    ... // обработка n прочитанных байтов  
}  
if (n == 0) {  
    // данных больше нет  
}  
else if (n < 0) {  
    if (errno == EINTR) {  
        // вызов был прерван, еще есть данные для чтения  
    }  
}
```


Прерывание системных вызовов

- в программах можно встретить конструкцию, подобную следующей:

```
do {  
    n = read(fd, buf, sizeof(buf));  
    // обработать прочитанные n байтов  
} while (n < 0 && IS_EINTR(errno));
```

- в glibc есть макрос TEMP_FAILURE_RETRY

```
int res = TEMP_FAILURE_RETRY(read(fd, buf, sizeof(buf)))
```

Обработка сигналов по требованиям POSIX

- ВВОДИТСЯ ПОНЯТИЕ маски сигналов (*signal mask*) типа `sigset_t`, для манипуляции с которой предназначены следующие функции:

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

```
int sigemptyset(sigset_t *set);
```

```
int sigfillset(sigset_t *set);
```

```
int sigaddset(sigset_t *set, int signum);
```

```
int sigdelset(sigset_t *set, int signum);
```

```
int sigismember(const sigset_t *set, int signum);
```

Установка маски сигналов

- установить новую маску и/или получить старую маску можно с помощью системного вызова `sigprocmask`:

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

```
int sigprocmask(
```

```
    int how, _____
```

```
    const sigset_t *set,
```

```
    sigset_t *oldset);
```

SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK

Определение наличия заблокированных сигналов

- для определения маски ожидающих доставки сигналов предназначен вызов ядра `sigpending()`:

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

```
int sigpending(sigset_t *set);
```

Установка обработчиков сигналов

- для установки обработчиков предназначен системный вызов `sigaction()`:

```
#include <signal.h>
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

Установка обработчиков сигналов

- для установки обработчиков предназначен системный вызов `sigaction()`:

```
struct sigaction {  
    void      (*sa_handler) (int);  
    void      (*sa_sigaction) (int, siginfo_t*, void*);  
    sigset_t   sa_mask;  
    int        sa_flags;  
    void      (*sa_restorer) (void);  
};
```

Расширенный обработчик

- устанавливается флагом SA_SIGINFO sa_flags

```
void SIG_handler(int signum,  
                 siginfo_t *info,  
                 void *ctx) {  
  
    . . .  
}
```

Сигналы реального времени

- SIGRTMIN .. SIGRTMAX
- гарантируется *надежная* доставка
 - сигналы ставятся в очередь
 - приоритет определяется номером сигнала (сначала доставляется сигнал с меньшим номером)
- могут передавать аргумент

```
union sigval {  
    int sival_int;  
    void *sival_ptr;  
};
```


Отправка сигнала реального времени

- для отправки обычно используется функция `sigqueue()`, хотя в Linux сработает и `kill`

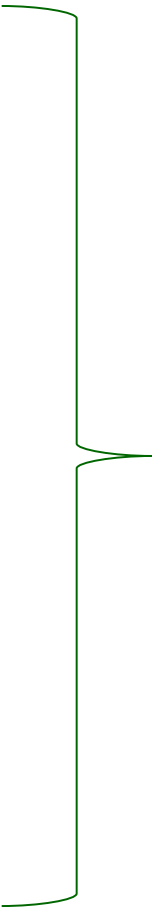
```
#include <signal.h>
```

```
int sigqueue(pid_t pid,  
             int sig,  
             const union sigval value);
```

Средства межпроцессного взаимодействия. Каналы

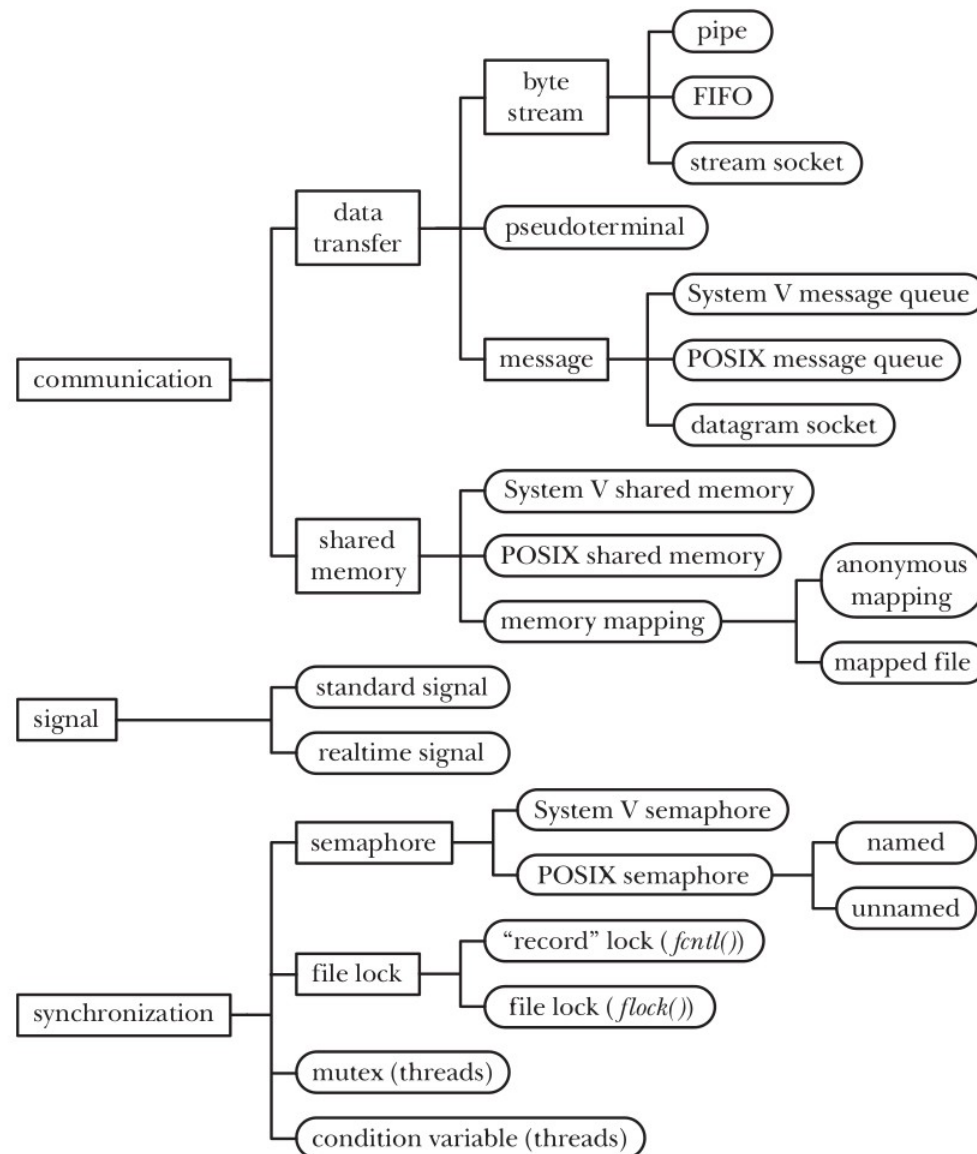
Средства межпроцессного взаимодействия

- Сигналы
 - реального времени
- Каналы
 - безымянные каналы
 - именованные каналы
- XSI IPC
 - Очереди сообщений
 - Семафоры
 - Разделяемая память
- Сокеты
 - локальные сокеты
 - сетевые сокеты



локальные IPC
(в пределах хоста)

Средства межпроцессного взаимодействия



Источник:
М.Kerrisk, TLPI

Безымянные каналы

```
#include <unistd.h>
```

```
int pipe(int pipefd[2]);
```

pipefd[0] – открыт для чтения

pipefd[1] – открыт для записи

Именованные каналы

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

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

```
int mkfifo(const char *pathname, mode_t mode);
```

Создание конвейеров

- оболочка (sh, bash, ...) позволяет запускать цепочки команд, связанных с помощью потоков ввода-вывода

```
> echo "hi there" | sed s/hi/hello/g  
hello there
```

Дублирование дескрипторов

```
#include <unistd.h>
```

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```


Управление атрибутами файла

- осуществляется с помощью системного вызова `fcntl()`

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ... );
```

Управление атрибутами файла

- дублирование дескриптора

...

```
int newfd = fcntl(olddfd, F_DUPFD, 0);
```

- перевод в неблокирующий режим

...

```
int flags = fcntl(fd, F_GETFL);  
flags |= O_NONBLOCK;  
fcntl(fd, F_SETFL, flags);
```

Средства межпроцессного взаимодействия. Разделяемая память

Разделяемая память

- **SysV** `shmget()`, `shmat()`, `shmctl()`
- **POSIX** `shm_open()`, `mmap()`, `shm_unlink()`

Отображение файлов в память

- для отображения файлов в память используется системный вызов `mmap()`

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

```
void *mmap(void *addr,  
           size_t length,  
           int prot,  
           int flags,  
           int fd,  
           off_t offset);
```

Отображение файлов в память

- для закрытия отображения используется системный вызов `munmap()`

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

```
int munmap(void *addr, size_t length);
```

Практический пример: доступ к
адресному пространству другого
процесса

Доступ к памяти других процессов

- в Linux список используемых процессом регионов в адресном пространстве может быть получен из файла `/proc/<pid>/maps`
- доступ к АП может быть выполнен
 - либо с помощью чтения/записи специального файла `/proc/<pid>/mem`
 - либо с помощью системных вызовов `process_vm_readv()` и `process_vm_writev()`

Доступ к памяти другого процесса как особый случай межпроцессного взаимодействия

- каким образом процесс, читающий/записывающий память другого процесса, может узнать адрес нужного региона памяти?
 - зарезервировать адрес (возможно ли наложение на разделяемые библиотеки? ASLR)
 - получить адрес
 - родственные процессы
 - через аргументы, env, ...
 - не родственники
 - через другое IPC
 - найти адрес
 - сканирование адресного пространства и поиск "маркера начала"

Средства межпроцессного взаимодействия. Сокеты

Сокеты

- Сокет – это средство межпроцессного взаимодействия, позволяющее взаимодействовать процессам как на одной, так и на разных машинах, находящихся в пределах компьютерной сети
- Семейства сокетов
 - локальные сокеты (UNIX domain)
 - сетевые сокеты (Internet domain)
 - IPv4
 - IPv6

Типы сокетов

- Типы сокетов
 - дейтаграммные (datagram)
 - потоковые (stream)
 - пакетные (seqpacket)
 - сырые (raw)

Характеристики сокетов

- Типы сокетов определяются набором характеристик, которыми обладает реализующий передачу данных протокол
 - ориентированность на соединение (connection oriented or connectionless)
 - сохранение границ сообщений или передача сплошным потоком (datagram or stream)
 - сохранение порядка доставки (delivery in sent order or out of sent order)
 - гарантия доставки (reliability)

Использование сокетов

- Сокет может использоваться в приложении в роли
 - клиентского сокета
 - серверного сокета
- Для некоторых типов сокетов такое разделение условно

Сокеты как точки коммуникации

- Сокет в приложении идентифицируется дескриптором, при создании сокета для него задается
 - семейство протоколов
 - тип (протокол)
 - адрес
 - для сетевых – ip/port
 - для локальных – имя файла

Сетевое взаимодействие

■ стеки сетевых протоколов

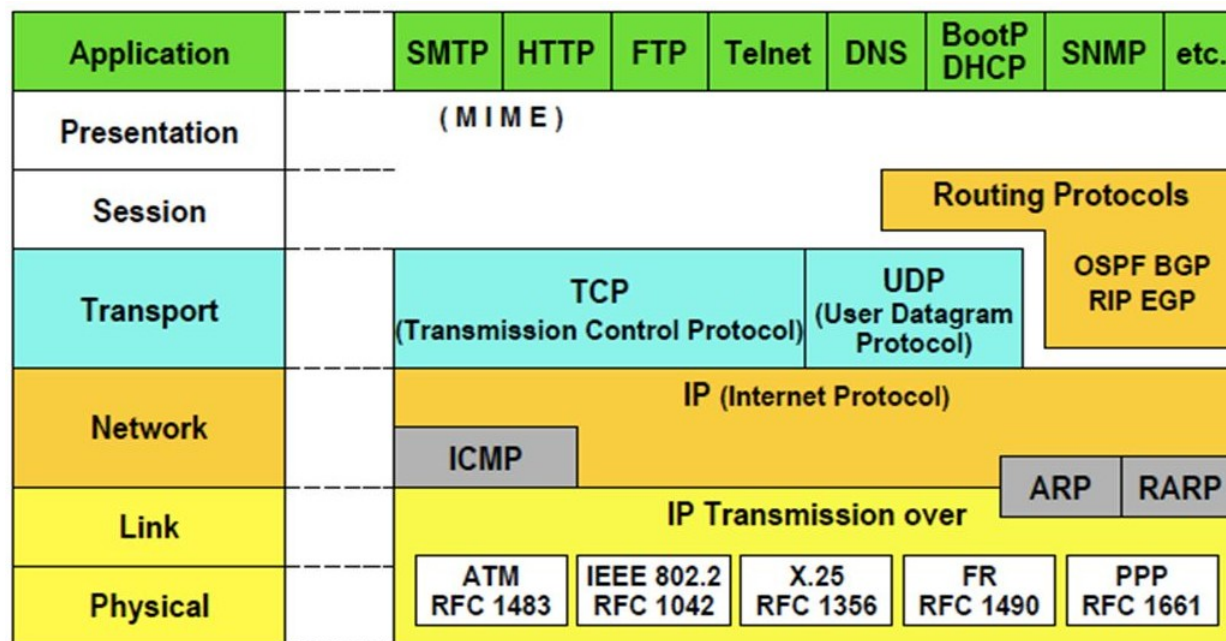
□ TCP/IP

■ IPv4

■ IPv6

□ IPX/SPX

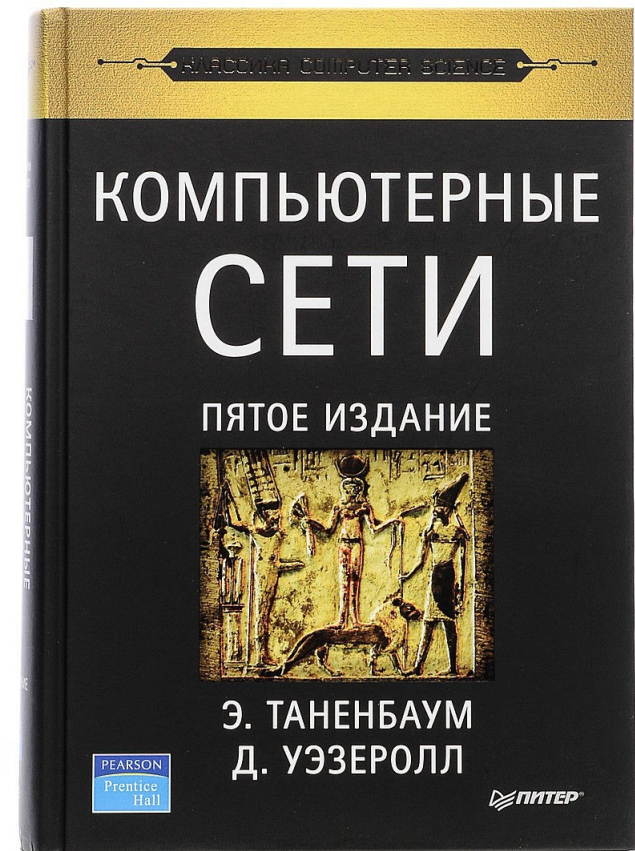
□ ...



семейство протоколов TCP/IP (IPv4)

Книги по теме

Таненбаум Э., Уэзеролл Д.
Компьютерные сети



Книги по теме

Олифер В.Г., Олифер Н.А.
Компьютерные сети

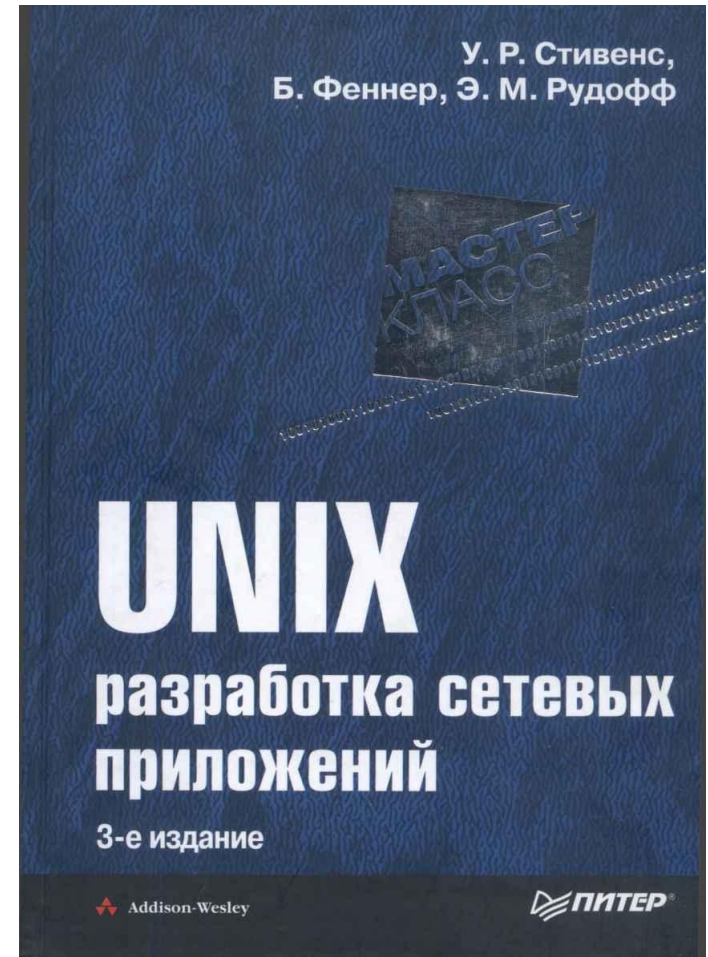


Книги по теме

Стивенс У.Р.

UNIX. Разработка сетевых
приложений

- главы
 - 3, 4, 8



Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type,
           int protocol);
```

Подключение сокета

```
#include <sys/types.h>
#include <sys/socket.h>

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

Структура адреса

```
#include <netinet/in.h>
```

```
struct sockaddr_in {  
    uint8_t            sin_len;  
    sa_family_t        sin_family;  
    in_port_t          sin_port;  
    struct in_addr      sin_addr;  
    char               sin_zero[8];  
};
```

```
struct in_addr {  
    in_addr_t    s_addr;  
};
```

Обобщенная структура адреса

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

struct sockaddr {
    uint8_t          sa_len;
    sa_family_t      sa_family;
    char             sa_data[14];
};
```

Значения для поля s_addr

```
struct sockaddr_in addr = {0};
```

```
...
```

```
addr.sin_addr.s_addr =  
    htonl(INADDR_LOOPBACK);
```

- INADDR_ANY
 - все адреса локального хоста (0.0.0.0)
- INADDR_LOOPBACK
 - адрес *loopback* интерфейса (127.0.0.1)
- INADDR_BROADCAST
 - широковещательный адрес (255.255.255.255)

Преобразование адресов

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

```
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
```

```
int inet_aton(const char *cp,  
             struct in_addr *inp);
```

```
in_addr_t inet_addr(const char *cp);
```

```
char *inet_ntoa(struct in_addr in);
```

Связывание сокета с адресом

```
#include <sys/types.h>
#include <sys/socket.h>

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

Перевод сокета в "прослушивающий" режим

```
#include <sys/types.h>
#include <sys/socket.h>

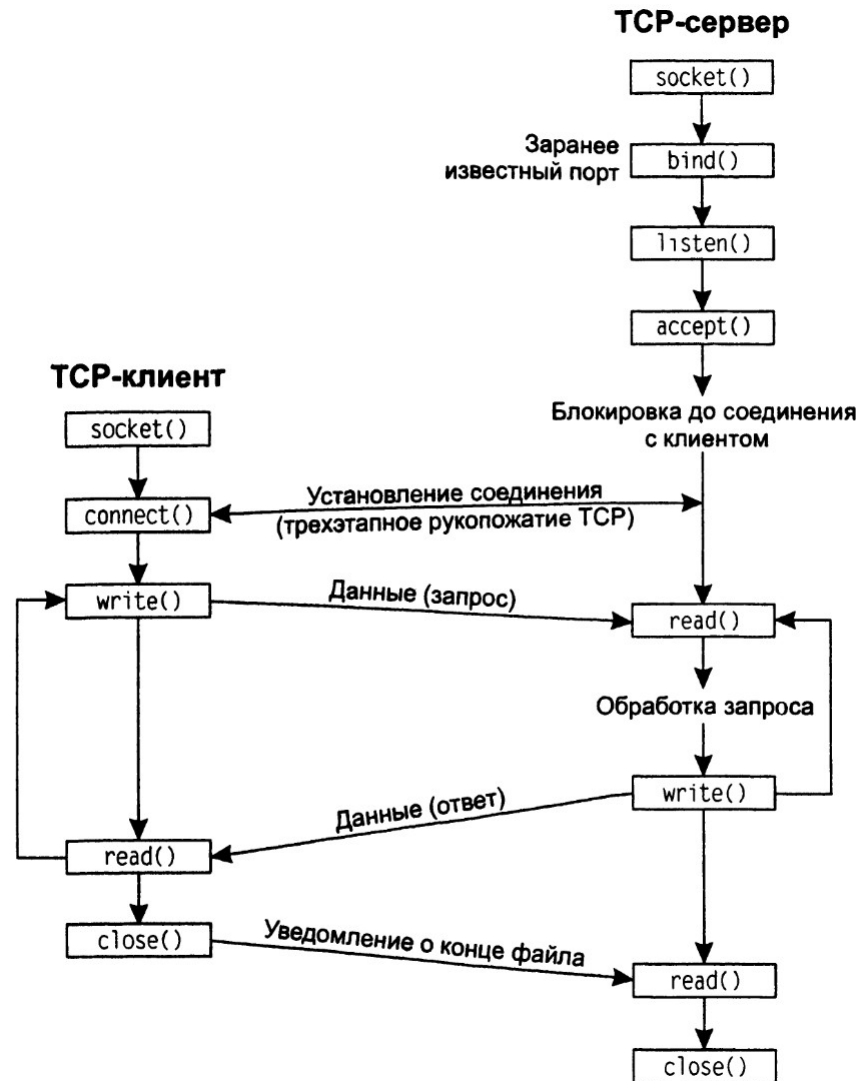
int listen(int sockfd, int backlog);
```

Ожидание входящих соединений

```
#include <sys/types.h>
#include <sys/socket.h>

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

Работа сервера и клиента TCP

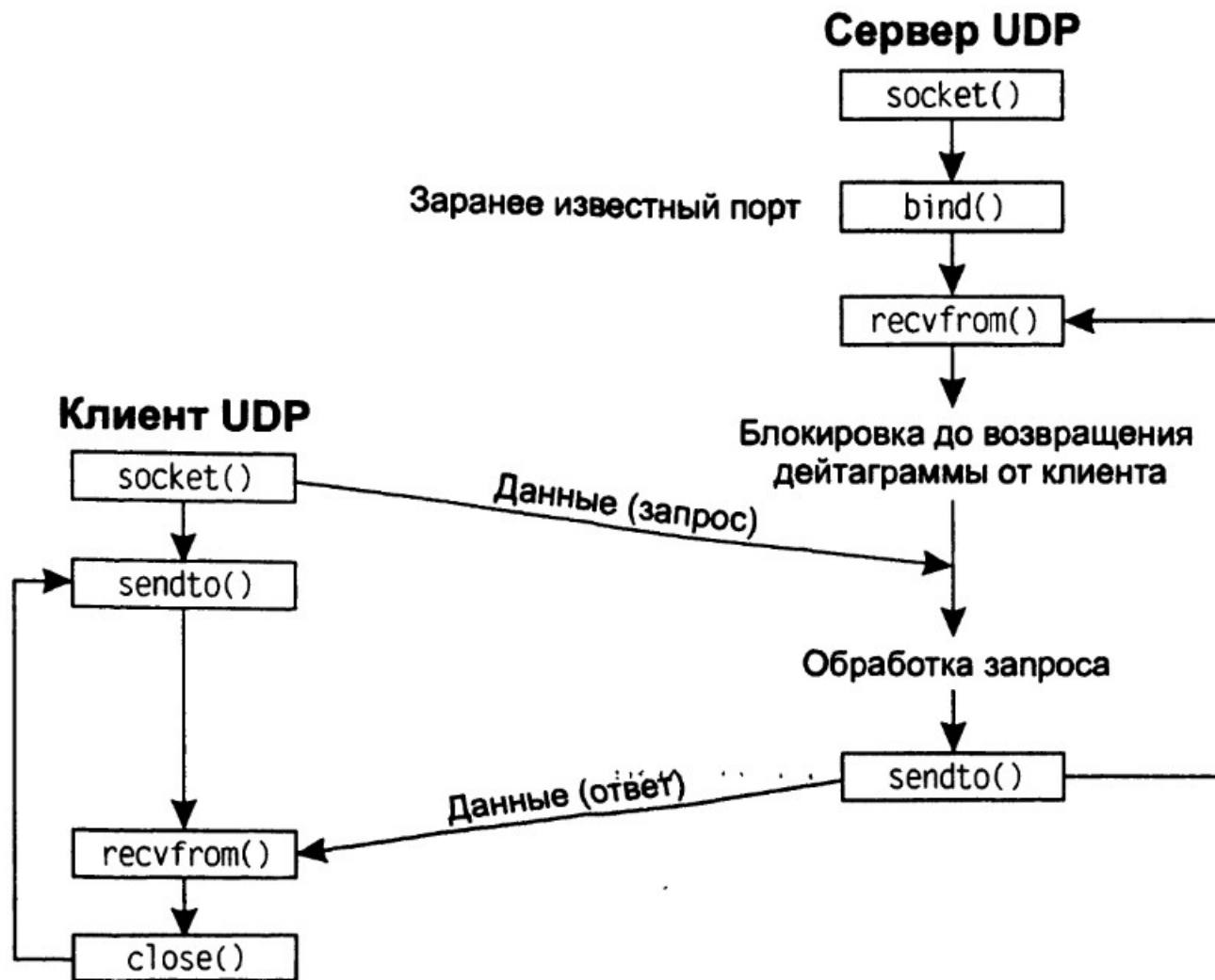


Получение информации о противоположной стороне

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

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

Работа сервера и клиента UDP



Отправка и получение данных через дейтаграммные сокеты

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

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

```
ssize_t sendto(int sockfd,  
               const void *buf, size_t len, int flags,  
               const struct sockaddr *dest_addr,  
               socklen_t addrlen);
```

```
ssize_t recvfrom(int sockfd,  
                 void *buf, size_t len, int flags,  
                 struct sockaddr *src_addr,  
                 socklen_t *addrlen);
```


Проверка ответа от сервера

```
int res = sendto(clientSocket, buffer,  
    strlen(buffer), 0,  
    (const struct sockaddr*)&serverAddr,  
    sizeof(serverAddr));
```

...

```
res = recvfrom(clientSocket, buffer,  
    sizeof(buffer), 0, replyAddr, &len);
```

```
if (len == sizeof(serverAddr) &&  
    memcmp(serverAddr, replyAddr, len) == 0) {  
    // reply from the server we sent request to  
}
```

Присоединенные и неприсоединенные сокеты UDP

- **неприсоединенный** (unconnected) сокет UDP – сокет, создаваемый по умолчанию
- **присоединенный** (connected) сокет UDP – сокет, для которого вызвана функция connect
 - для него нет необходимости задавать адрес назначения
 - можно использовать send()/write() вместо sendto()
 - можно использовать recv()/read() вместо recvfrom()
 - позволяет получить ошибку доставки (при чтении ответа)
 - позволяет не получать датаграммы, отправленные с других хостов (кроме того, к которому выполнено "присоединение")

Вопрос о размере поступивших дейтаграмм

- если используется UDP, то максимальный размер дейтаграммы немного меньше 64Kb
 - нужно также учитывать MTU
 - если приемный буфер в приложении меньше, чем поступившая дейтаграмма, данные, которые в него не удастся скопировать, будут потеряны

Вопрос о размере поступивших дейтаграмм

- как правило, максимальный размер сообщения известен заранее (определяется прикладным протоколом)
 - поэтому можно просто использовать буфер, рассчитанный на прием самого большого сообщения
- если нужно узнать размер пришедшей дейтаграммы/всех данных в буфере ядра, то можно использовать `ioctl()` с `FIONREAD`

Задание на дом

- читать
 - Стивенс, Раго. UNIX. Профессиональное программирование
 - главы 10, 15, 16
 - Лав. Linux: Системное программирование
 - глава 10
 - глава 4
- раздел “Отображение файлов в память”

