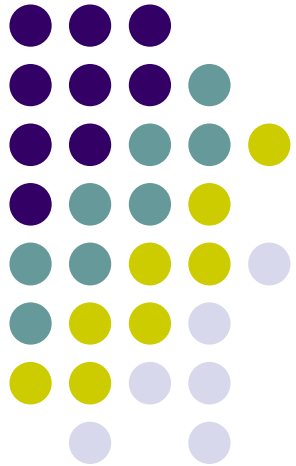
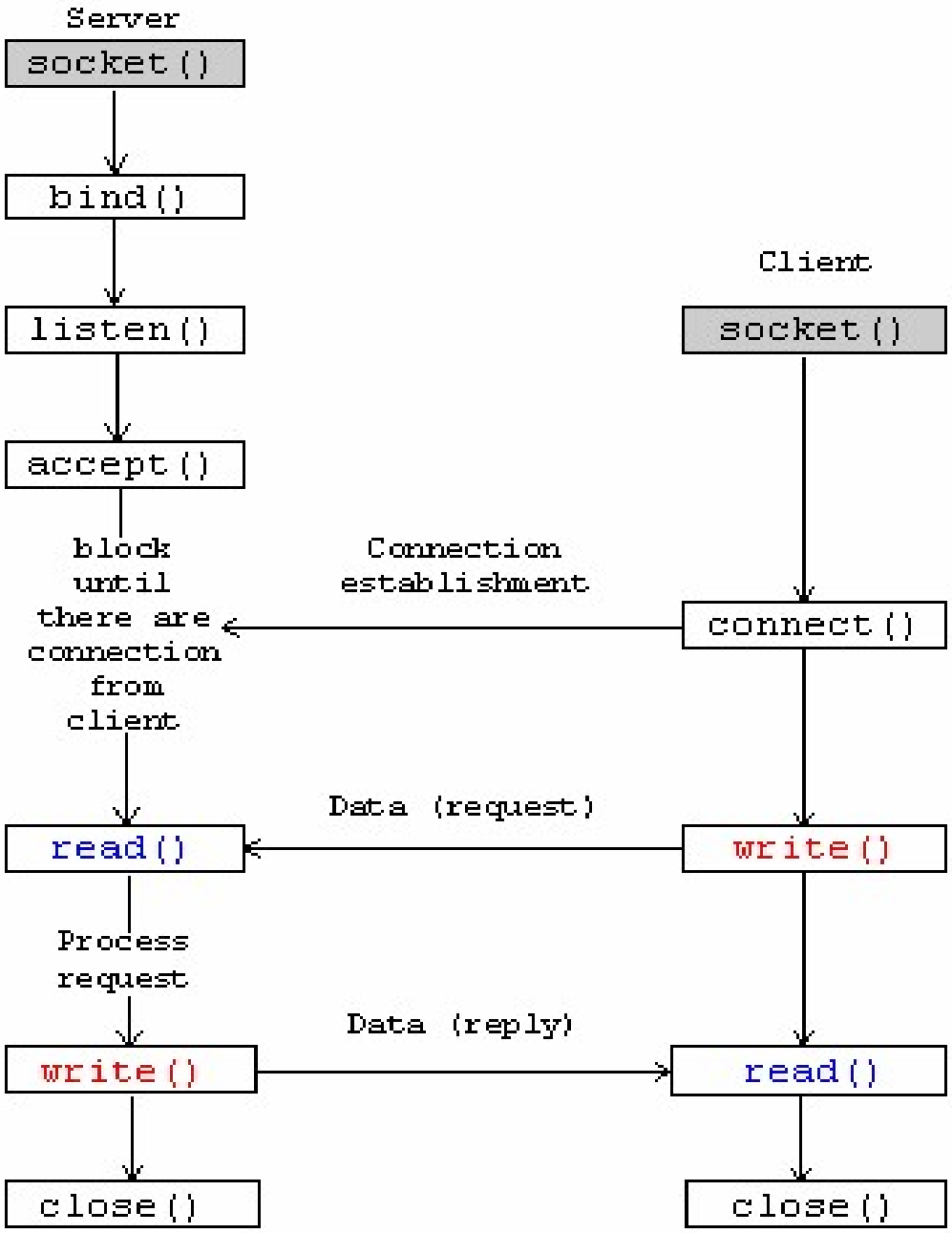
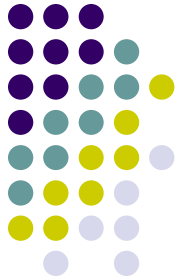
Мрежово програмиране

TCP сокети




Server
socket()
bind()
listen()
accept()
block until there are connection from client
Client
socket()
Connection establishment
connect()
Data (request)
read()
write()
Process request
Data (reply)
write()
read()
close()
close()

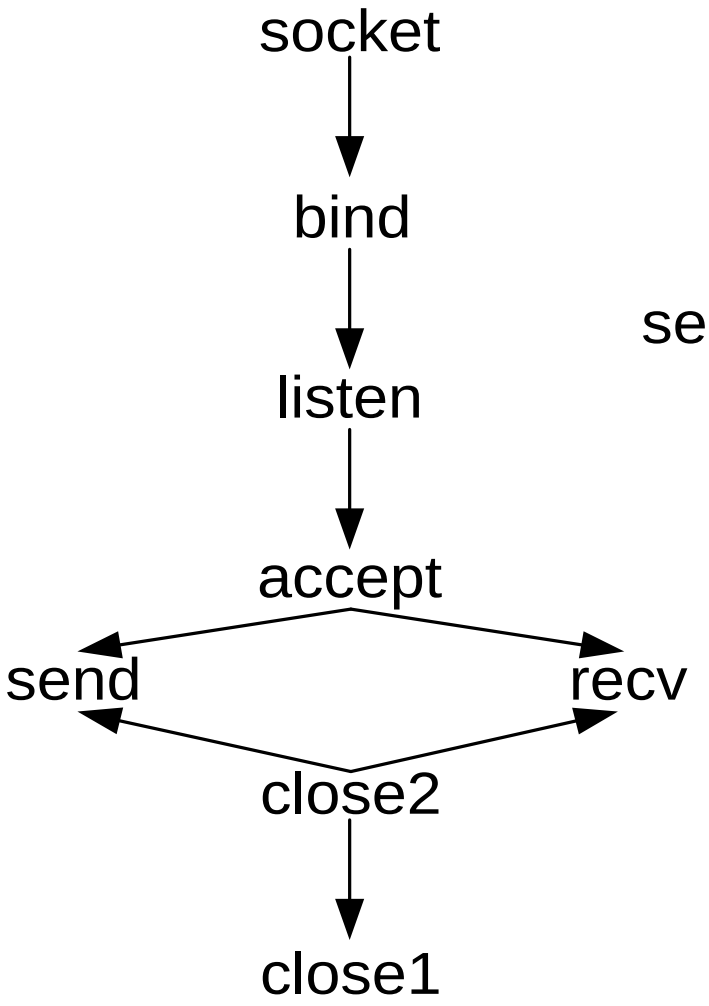


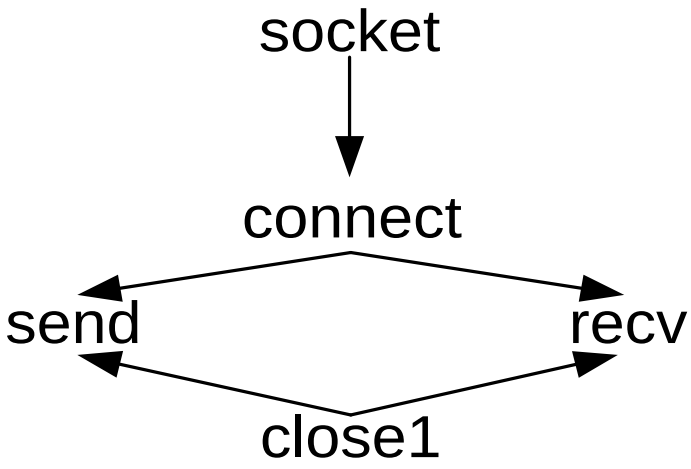
Действия на сървъра:

- Създаване на сокет
- Именоване на сокета
- Привеждане на сокета в режим на прослушване
- Приемане на съединение (създаване на нов сокет)
- Обслужване на клиента
- Затваряне на сокетите

Действия на клиента:

- Създаване на сокет
- Иницииране на съединение със сървъра
- Изпращане на заявка към сървъра
- Приемане на отговора
- Затваряне на сокета


socket
bind
listen
accept
send
recv
close2
close1


socket
connect
send
recv
close1

Нови примитиви

- **connect()**

1. Свързва сокета към IP адрес и номер на порт
2. Изпраща заявка за установяване на съединение и установява съединение

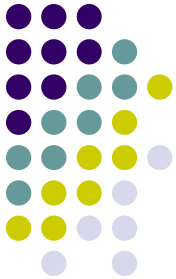
- **listen()**

1. Привежда сокета в пасивно състояние
2. Създава опашки за съединения

- **accept()**

1. Очаква желаещите да установят съединения
2. Установява съединение
3. Създава нов потоков сокет

Установяване на логическо съединение на клиенстката страна. Системен примитив connect()



- Установява лог. съединение със сървъра.
- Извикването connect() скрива вътре в себе си настройката на сокета на избран системен порт и произволен мрежов интерфейс (на практика това е примитивът bind() с нулев номер на порт и IP адрес INADDR_ANY).
- Примитивът се блокира докато не бъде установено логическо съединение или не изтече определен интервал от време, който може да се регулира от системния администратор.

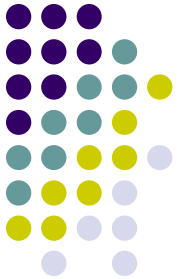
Прототип на системния примитив

```
#include <sys/types.h>
#include <sys/socket.h>
    int connect(int sockd,            /*дескриптор*/
    struct sockaddr *servaddr,        /*адрес*/
    int addrlen);                     /*дължина на структурата*/
```

Връща: 0 – при успешно установено логическо съединение, -1 – при грешка.

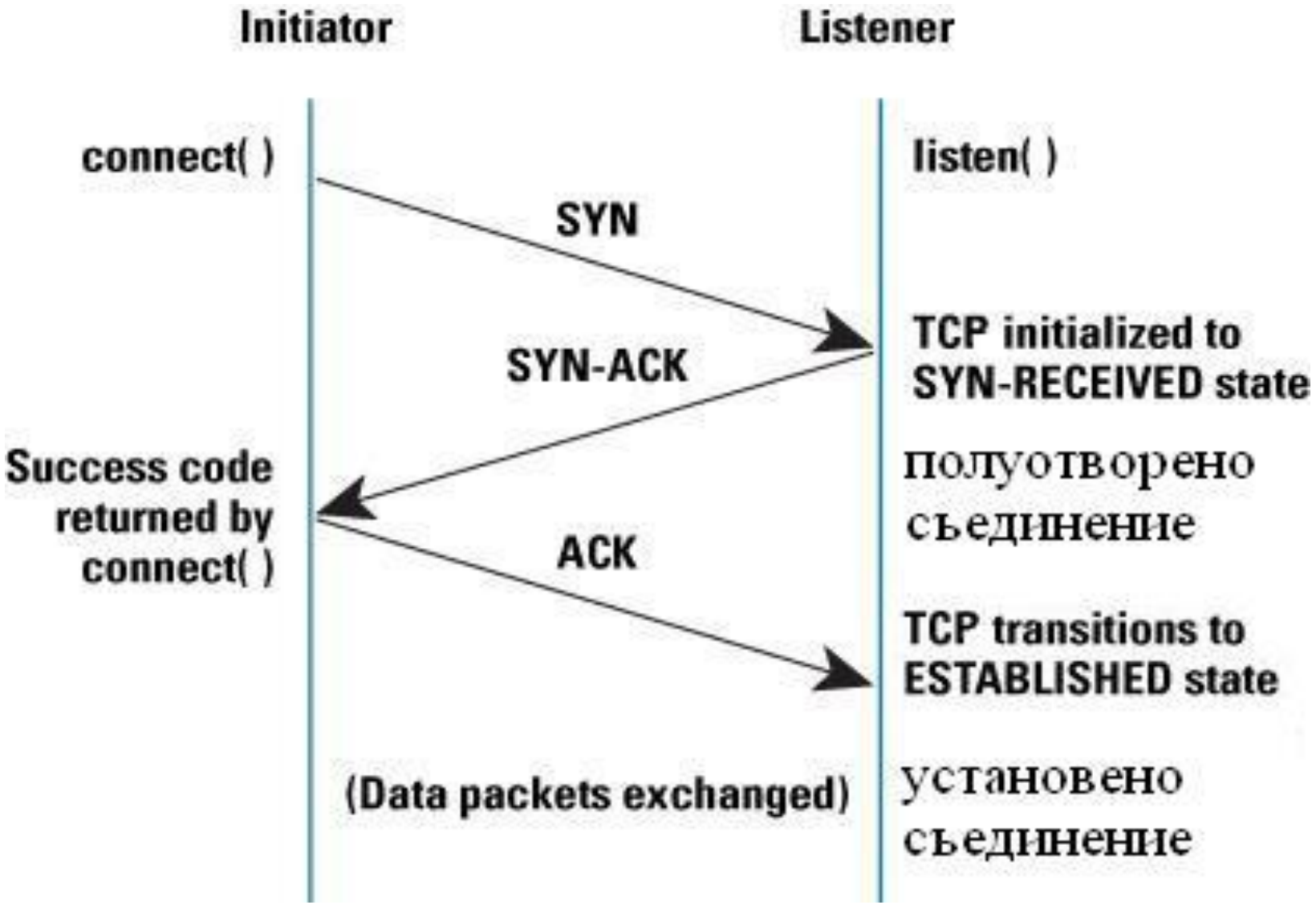
Вторият параметър е указател към структура от данни с адреса на сокет, съдържаща адреса на отдалечения сокет. Ако сокетът *sockd* няма локално име, тогава този примитив го задава.
Примитивът може да се блокира.
Ако установяването на съединението не е успешно, тогава следва да се затвори неуспешно използвания сокет и отново да се извика функцията socket().

Как се реализира виртуално съединение



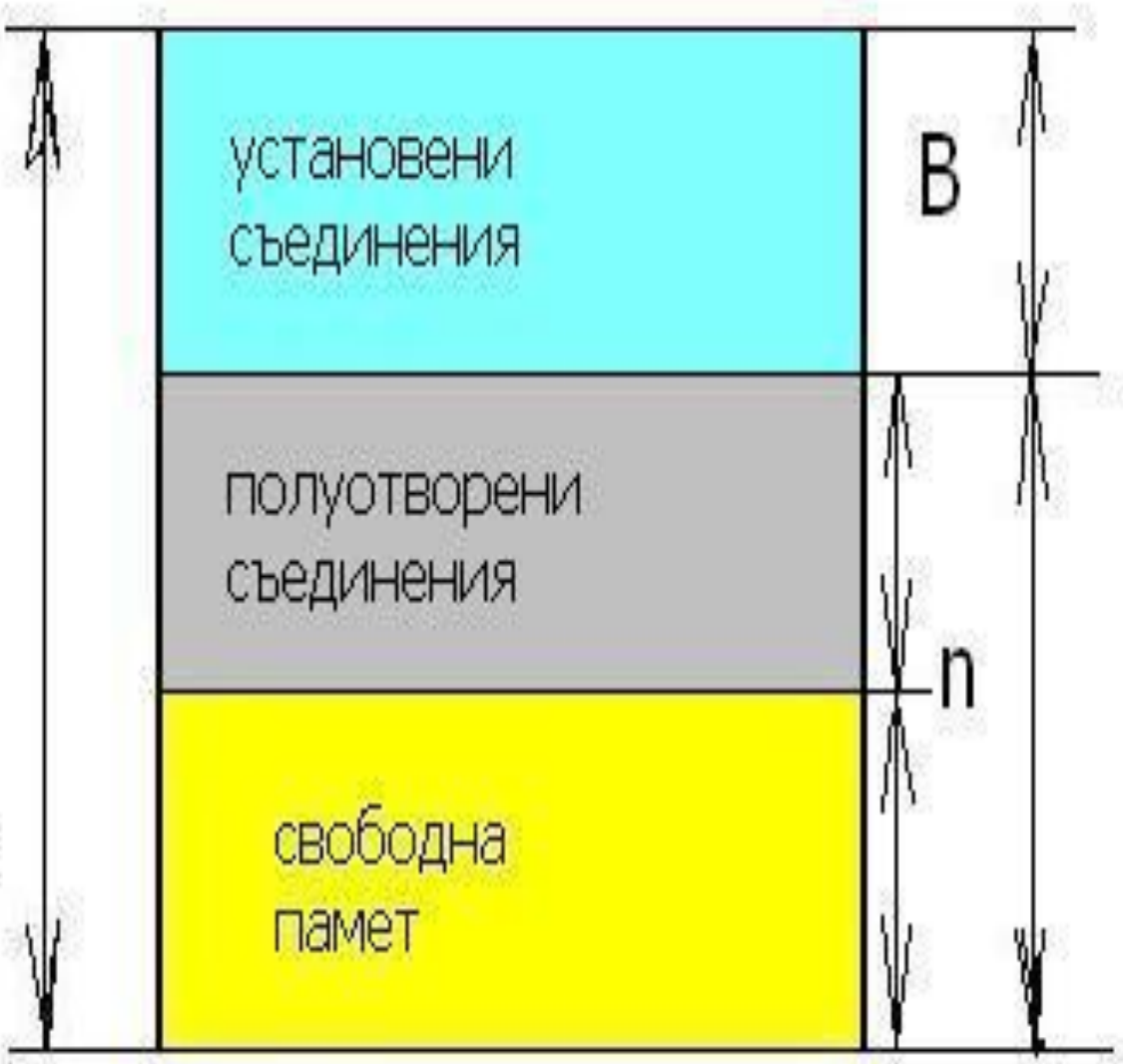
- TCP протоколът е надежден дуплексен протокол.
- В TCP протокол се използва:
  - номериране на предаваните пакети;
  - контрол на реда на получаването им;
  - потвърждения за получаването на пакета;
  - повторно изпращане при отсъствие потвърждение;
  - изчисляване на контролни суми за предаваната информация.

Три етапно ръкостискане (three-way handshake)

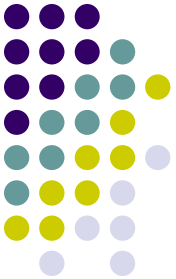


L -

буфер
за
съхранение
на
параметрите
на TCP
съединението



Установяване на логическо съединение на сървърната страна



- Създава се сокет и се настройва към локалния адрес.
- Следва създаване на опашка от заявки за съединения (listen).
- Когато сървърът е готов да обслужи поредната заявка, той използва accept.
- accept създава за комуникация с клиента нов сокет и връща неговия дескриптор.

Системен примитив listen()

Прототип на системния примитив:

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Параметри:

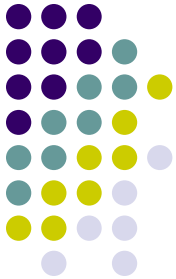
- *sockd*: дескриптор на сокета;
- *backlog*: максимална дължина на опашката за съединенията, очакващи потвърждение (за много ОС не може да превишава 5; в GNU Linux се предлага стойността SOMAXCONN=128).

Резултат:

- привежда сокета в пасивен режим;
- установява дълбочината на опашката за съединения (броя на съединенията).

Връща: 0 – при успешно приключване, -1 – при грешка.

Системен примитив accept()



Прототип на системния примитив:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd,                /*дескриптор*/
    struct sockaddr *cliaddr,        /*адрес*/
    int *clilen);                    /*дължина на адреса*/
```

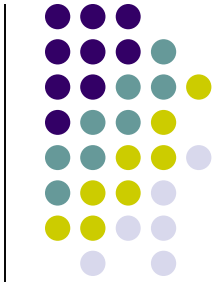
- използва се от сървъра за приемане на напълно установеното съединение;

  Връща: -1 – при грешка, иначе – цяло неотрицателно число (*дескриптор на новия сокет*). Новият сокет не се намира в състояние на прослушване, атрибутите на сокета *sockd* (например флаг O_NONBLOCK – установяване на сокет за неблокиращ вход-изход или на флаг O_ASYNC – установяване на сокет за вход-изход, управляем чрез сигнал) не се наследяват. Състоянието на изходния сокет *sockd* не се променя. Примитивът може да бъде блокиращ.

Потоково ориентираните сокети се делят на активни и пасивни

- Активният сокет е съединен с отдалечен активен сокет, като използва установено съединение за данни.
- Затварянето на съединението унищожава активните сокети и в двете точки на съединението.
- Пасивният сокет с никого не е съединен. Той чака заявка за съединение.

Затваряне на сокет



След приключване на обмена на данните програмата трябва да затвори сокетите като извика функцията close().
Прототипът на тази функция е описан във файла unistd.h и има вида:

int **close**(int fd);

На тази функция се предава дескриптора на сокета, който трябва да се затвори.
При успешно приключване функцията връща 0, при грешка - -1.

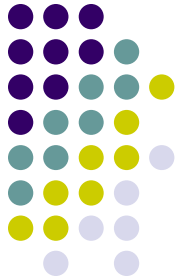
Може да се забрани предаването на данни в едно от двете (или и двете) направления с shutdown

int shutdown(int sockfd, int how);

Параметърът how може да приема следните стойности:

- 0 – да се забрани четене от сокета
- 1 – да се забрани записване в сокета
- 2 – да се забранят и двете

Въпреки, че след извикване на shutdown с параметър how, равен на 2, вие повече не можете да използвате сокета за обмен на данни, все пак се налага да извикате close, за да освободите свързаните с тях системни ресурси.



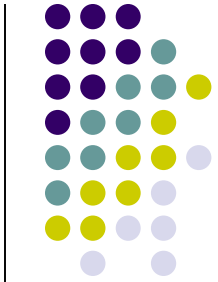
Някаква бъркотия внасят присъединените дейтаграмни сокети (connected datagram sockets).

За сокет от тип SOCK_DGRAM също може да се извика connect, а след това да се използват send и recv за обмен на данни.

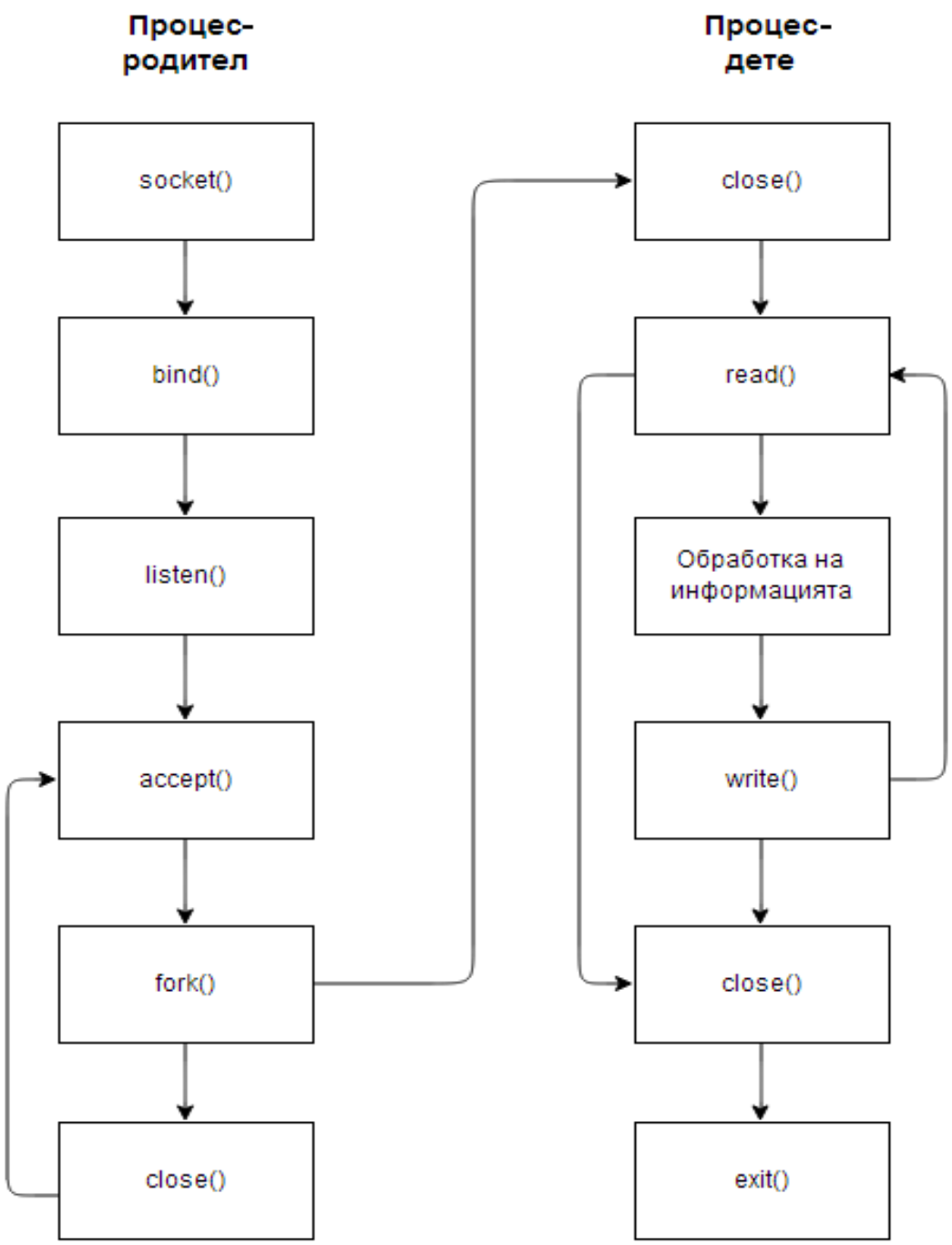
Необходимо е да се знае, че никакво логическо съединение при това не се установява. Операционната система просто запомня адреса, който е получен благодарение на използването на connect, а след това го използва за изпращане на данните.

Обърнете внимание, че присъединеният сокет може да получава данни само от сокет, с който е реализирано логическо съединение.

Създаване на програми за паралелна обработка на заявките на клиентите



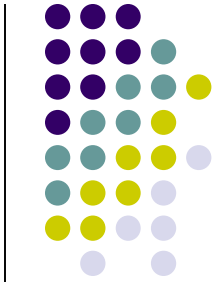
- Схема на работа на TCP сървър с паралелна обработка на заявките
  - Примитивът listen не само привежда сокета в пасивен режим на очакване на клиентска заявка, но също така го подготвя за обработка на множество едновременно постъпващи заявки.


Процес-
родител
Процес-
дете
socket()
bind()
listen()
accept()
fork()
close()
close()
read()
Обработка на
информацията
write()
close()
exit()

Конструкцията на сървъра за паралелна обработка предполага, че за всяка нова заявка от клиента се създава ново копие на сървърния процес

- След като се е изпълнил примитивът accept, паралелният сървър ще създаде нов (породен) процес и ще му предаде задачата за обслужването на новата заявка.
  - В родителския процес върнатата стойност съдържа идентификатора на породения процес, а в породения - тази стойност е равна на нула.

Работа с процеси



Създаване на процес–дете

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

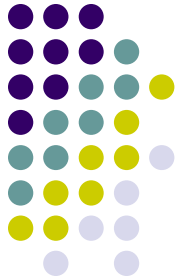
При грешка функцията връща -1, при което не се създава процес-дете.
В родителския процес функцията връща PID на новия процес.
В процеса-дете тази функция връща 0.

Пример:

```
pid_t new_pid;
new_pid=fork();
if (new_pid == -1) { /* грешка */ }
if (new_pid == 0) {
   /* процес-дете */
}
else {
   /* родителски процес */
}
```

Принципи за проектиране на паралелен сървър

- Сървърът за паралелна обработка създава нов процес за всяко съединение. С други думи, на хоста постоянно работи единствен главен сървър, очаквайки заявка от някакъв процес в мрежата.
- В друг момент от време, на същия този хост както преди работи главен сървър, а заедно с него и множество подчинени. Всеки подчинен сървър работи с уникален адрес на конкретния, съединен с него процес.



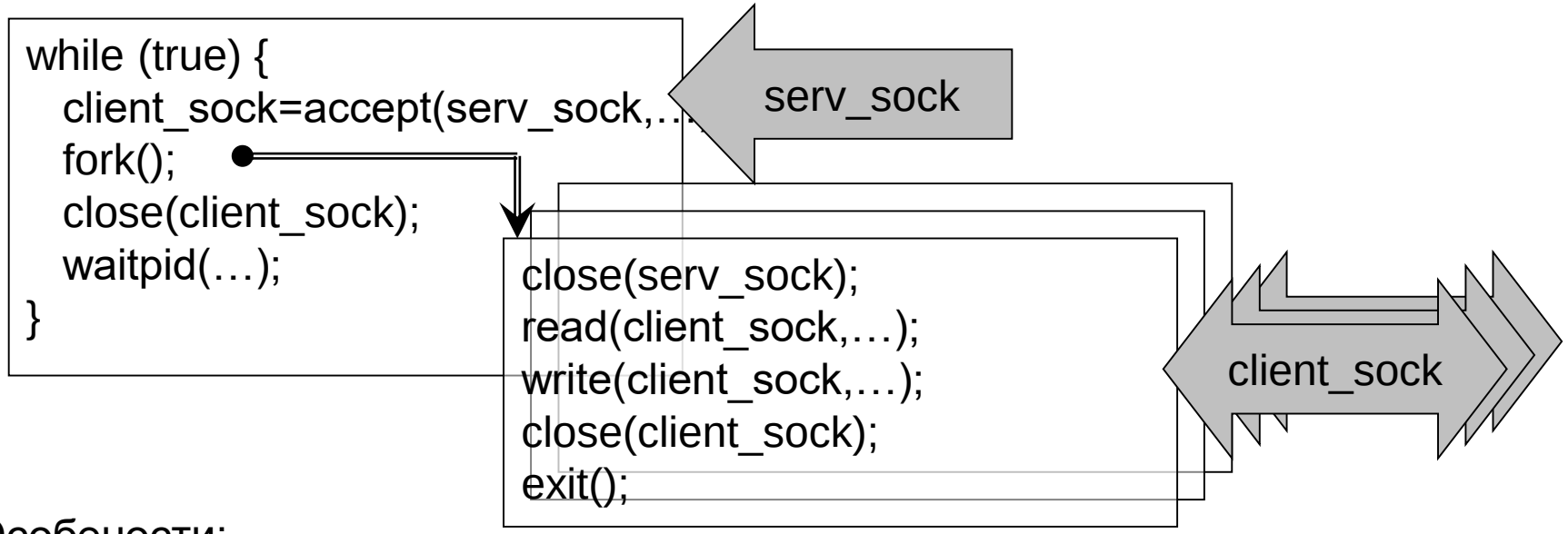
Предимството на този подход е в това, че така се пишат доста компактни и разбираеми за четене програми, в които кодът за установяване на логическо съединение е отделен от кода за обслужване на клиентите.

За съжаление, този подход има и недостатъци:

1. Ако клиентите са твърде много, създаването на всеки нов процес за обслужване на всеки от тях може да се окаже скъпо «удоволствие»;
2. Този подход неявно предполага, че всички клиенти се обслужват независимо един от друг. Например ако пишете чат – сървър, трябва да се поддържа взаимодействието на всички клиенти, които са се присъединили към сървъра. В тези условия границите между процесите ще са бариера и ще трябва да се избере друг подход за обслужване на клиентите.

Синхронен вход/изход

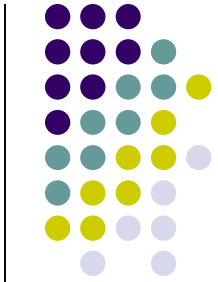
Обслужването на всеки дескриптор на файла представлява отделен процес (отделна нишка).



Особености:

- не се изисква мултиплексиране на I/O
- проблеми със синхронизацията и взаимодействието
- изисква мощни ресурси на сървъра
- възможна е паралелна обработка на заявките

С функцията fcntl - превръщаме сокета в неблокиращ

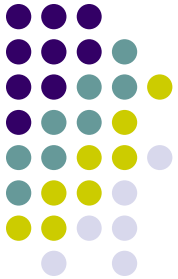


Например, ако сме извикали read, а данни на нашия край на съединението няма, тогава в очакване на пристигането им нашата програма "заспива". Аналогична е ситуацията, когато извикваме accept, а опашката със заявките за съединение е празна.

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

Функцията fcntl изпълнява определени манипулации с файловия дескриптор. Тук TCP сокетът е обявен за неблокиращ с командата F_SETFL с флаг O_NONBLOCK.

Тази несложна операция превръща сокета в неблокиращ. Извикването на коя да е функция с такъв сокет ще връща управлението незабавно.

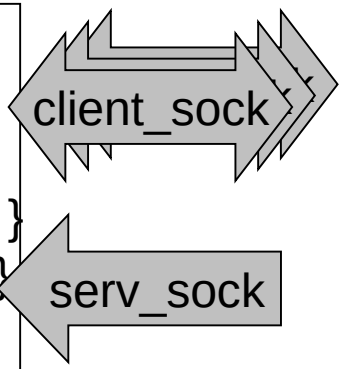


- При което ако операцията не е била изпълнена докрай, функцията ще върне -1 и ще запише в errno стойността EWOULDBLOCK.
- За да дочакаме приключването на операцията, можем да проверяваме всички наши сокети в цикъл, докато някоя функция не върне стойност, различна от EWOULDBLOCK.
- Когато това се случи, вече ще можем да стартираме за изпълнение следваща операция с този сокет и да се върнем към проверяващия цикъл.
- Такава тактика (polling) е работоспособна, но не е ефективна, понеже процесорното време се харчи напразно за многократни (и безрезултатни) проверки.

Неблокиращ вход/изход

Системното извикване приключва веднага с код за грешка EAGAIN (в BSD – EWOULDBLOCK). Необходимо е периодично да се повторя извикването до тогава, докато не приключи успешно или не възникне друга грешка.

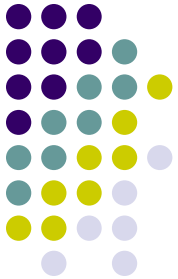
```
i = 0;
while (true) {
   len = read(client_socks[i],…);
   if (len == 0) { /* съединението е затворено */ }
   if (len < 0 && errno != EAGAIN) { /* грешка */ }
   if (len > 0) serve_client(i);
   new_sock = accept(serv_sock,…);
   if (new_sock >= 0) {
       fcntl(new_sock, F_SETFL, O_NONBLOCK);
       /* добавяме нов сокет в client_socks[] */
   }
   …
   i = ++i  % num_socks;
}
```



Особености:

- празни цикли → намалява производителността на системата
- едновременно се обслужва само 1 клиент

С цел да се подобри ситуацията се използва функцията select



- **select** (nonblocking sockets) често се използва от сървърите.
- Сложните клиентски програми също така могат да я използват.
- Тази функция позволява на единичен процес да следи едновременно за състоянието на няколко файлови дескриптора (а в Unix тук са и на сокетите).

Мултиплексиране на въвеждането и извеждането

Синхронно мултиплексиране на вход/изход

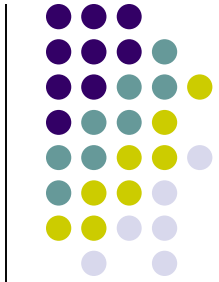
```
#include <sys/select.h> /* POSIX 1003.1-2001 */
int select(
        int n,                  /*макс. брой дескриптори*/
        fd_set *readfds,        /*дескриптори за четене*/
        fd_set *writefds,       /*дескриптори за записване*/
        fd_set *exceptfds,      /*дескриптори за изключения*/
        struct timeval *timeout);  /*таймаут*/
```

При грешка функцията връща -1. В случай на успех функцията връща броя на дескрипторите, готови за изпълнение на операции без блокиране. Функцията връща 0, ако е изтекло времето за очакване (*timeout*).

Възможни грешки:

EBADF – в множеството е включен невалиден дескриптор
EINTR – извикването е било прекъснато от сигнал
EINVAL – некоректни стойности за *n* или *timeout*
ENOMEM – недостиг на памет за изпълнение на операцията

Мултиплексиране (2)

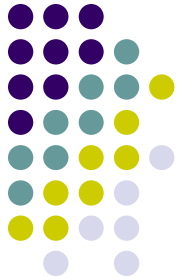


Преди да се извика select() е необходимо да се попълнят битовите полета на fd_set. При успешен възврат от select() в тези битови полета остават дескрипторите, за които:
readfds: възможно е четене без блокиране, в това число и на нула байта (EOF); за сокети в режим на прослушване – дали е възможен accept() без блокиране;
writefds: възможно е записване без блокиране; за сокети за които е установено съединение (connect);
exceptfds: за сокетите с пакети с флаг URG (out-of-band data).

Битовото поле fd_set може да има максимална дължина FD_SETSIZE. Дескрипторите с номера по-големи от FD_SETSIZE не могат да бъдат записани в битовото поле FD_SETSIZE (обикновено FD_SETSIZE=1024).

Аргументът *n* задава максималния брой на дескрипторите + 1. Обикновено в качеството на този параметър се записва стойността FD_SETSIZE.

Мултиплексиране (3)



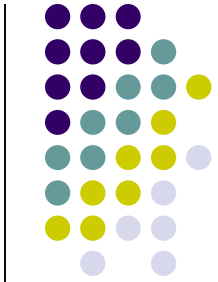
timeout:
Ако *timeout е* NULL, тогава select() може да се блокира за неопределено време – до «готовност» на един от зададените в множествата дескриптори или до получаването на сигнал.

```
#include <sys/types.h> /* или <sys/time.h> */
struct timeval {
        long tv_sec;  /* секунди */
        long tv_usec; /* микросекунди */
};
```

Ако *timeout.tv_sec*=0 и *timeout.tv_usec*=0, тогава select() приключва веднага и информира дали има «готови» дескриптори.

В GNU Linux при възврат от select() *timeout* съдържа останалото до изтичането на таймаут-а време. В другите системи такова поведение не е предвидено. POSIX препоръчва да се счита, че при възврат от select() стойността на *timeout* не е определена (т. е. тя не може да се използва повторно).

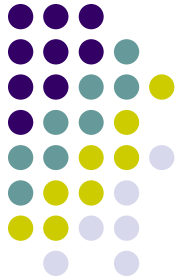
Мултиплексиране (4)



Манипулиране с битови полета на fd_set

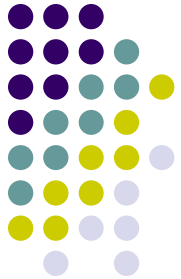
```
#include <sys/select.h> /* POSIX 1003.1-2001 */

/* почиства множеството fds: */
void FD_ZERO(fd_set *fds);

/* добавя дескриптора fd в множеството: */
void FD_SET(int fd, fd_set *fds);

/* изтрива дескриптора fd от множеството: */
void FD_CLR(int fd, fd_set *fds);

/* проверява дали се съдържа в множеството дескриптора fd: */
int FD_ISSET(int fd, const fd_set *fds);
```

Мултиплексиране (5)



Пример:

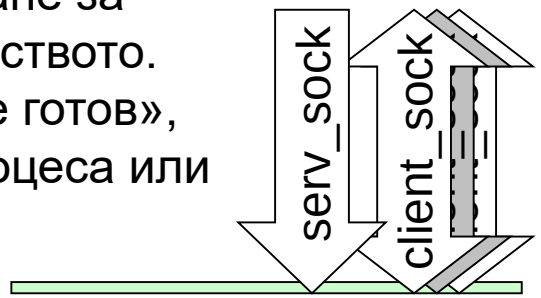
```
fd_set rfds;
struct timeval tv;
int n, i;
while (1) {
  FD_ZERO(&rfds);
  for (i = 0; i < num; i++) FD_SET(fd[i], &rfds);
  tv.tv_sec = 5;     /* изчакване 5 секунди */
  tv.tv_usec = 0;
  n = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
  if (n == -1) { /* грешка */};
  else if (n > 0) {
    for (i=0; i<num; i++) if (FD_ISSET(fd[i], &rfds)) {
      /* реализация на вход-изход */
    }
  }
  else if (n == 0) { /* изтекъл е таймаут-а */ }
}
```



- Ако поне един сокет е готов за изпълнение на зададената операция, тогава select връща ненулева стойност, а всички дескриптори, довели до сработването на функцията се записват в съответните множества.
- Това ни позволява да анализираме съдържащите се в множествата дескриптори и да изпълним с тях необходимите действия.

I/O с възможност за уведомяване за състояние на готовност

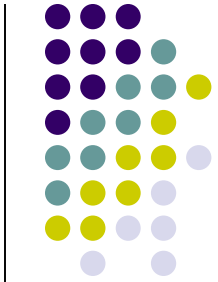
Процесът запитва ядрото на ОС дали е възможен вход/изход без блокиране за някой от дескрипторите от множеството. Ако никой от дескрипторите «не е готов», извикването може да блокира процеса или да приключи веднага.




```
while (true) {
    /* подготовка за проверка */
    num = select(…); /* или poll(…) */
    if (num > 0) {
        /* преминаване през множеството
от дескриптори с обслужване на «готовите» */
    }
}
```

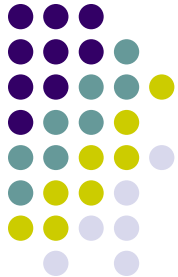
Особености:

- ограничение FD_SETSIZE за select()
- едновременно се обслужва само един клиент



- Програмите, използващи неблокиращи сокети заедно с функцията select, се получават доста объркани.
- Ако в случая с fork ние построяваме логиката на програмата така, като че ли клиентът е единствен, тук програмата е принудена да проследява дескрипторите на всички клиенти и да работи с тях паралелно.

Асинхронен вход/изход



Операциите за вход/изход веднага връщат процеса, който ги е извикал. За приключването на операцията ОС уведомява процеса чрез сигнал (в GNU Linux; съобщение – в MSWindows) или чрез специален сигнализиращ обект (MSWindows).

В GNU Linux интерфейсът AIO не позволява да се работи със сокети.

В MSWindows сокетите и файловете имат различни механизми за асинхронен вход/изход. За файлове и обекти, отваряни с помощта на CreateFile(), асинхронния вход/изход се реализира с помощта на Overlapped I/O, за сокети е реализиран Winsock във вид на отделен набор от функции.

Особености:

- проблеми със синхронизацията на достъпа до данните;
- сложности с дебъгването;
- едновременно се обслужва само един клиент.